

Handbook of Magma Functions

Part V: Lattices and Quadratic Forms

Version 2.30-1

Sydney

November 10, 2026

Editors

John Cannon *Wieb Bosma* *Claus Fieker* *Allan Steel*

Handbook Contributors

Eran Assaf *Geoff Bailey* *Wieb Bosma* *Gavin Brown* *Nils Bruin* *John Cannon* *Jon Carlson* *Scott Contini* *Edgar Costa* *Bruce Cox* *Brendan Creutz* *Steve Donnelly* *Tim Dokchitser* *Willem de Graaf* *Andreas-Stephan Elsenhans* *Claus Fieker* *Damien Fisher* *Volker Gebhardt* *Sergei Haller* *Michael Harrison* *Florian Hess* *Derek Holt* *David Howden* *Al Kasprzyk* *Markus Kirschmer* *David Kohel* *Axel Kohnert* *Dimitri Leemans* *Paulette Lieby* *Graham Matthews* *Scott Murray* *Eamonn O'Brien* *Dan Roozemond* *Ben Smith* *Bernd Souvignier* *William Stein* *Allan Steel* *Damien Stehlé* *Nicole Sutherland* *Don Taylor* *Bill Unger* *Alexa van der Waall* *Paul van Wamelen* *Helena Verrill* *John Voight* *Mark Watkins* *Greg White*

Production Editors

Wieb Bosma *Claus Fieker* *Allan Steel* *Nicole Sutherland*

HTML Production

Claus Fieker *Allan Steel*

Copyright © 1994–2026 The University of Sydney. All rights reserved.

Contents

31 Lattices **1**
- 31.1 Introduction . . . 1
- 31.2 Presentation of Lattices . . . 1
- 31.3 Creation of Lattices . . . 2
- 31.4 Lattice Elements . . . 9
- 31.5 Properties of Lattices . . . 12
- 31.6 Construction of New Lattices . . . 14
- 31.7 Reduction of Matrices and Lattices . . . 18
- 31.8 Minima and Element Enumeration . . . 30
- 31.9 Theta Series as Modular Forms . . . 44
- 31.10 Voronoi Cells, Holes, and Covering Radius . . . 45
- 31.11 Orthogonalization . . . 47
- 31.12 Testing Matrices for Definiteness . . . 48
- 31.13 Genera and Spinor Genera . . . 49
- 31.14 Attributes of Lattices . . . 56
- 31.15 Database of Lattices . . . 56
- 31.16 Bibliography . . . 61

32 Lattices over Number Fields **63**
- 32.1 Introduction . . . 63
- 32.2 Number Field Lattices . . . 63
- 32.3 Number Field Lattice Elements . . . 78
- 32.4 Examples . . . 81
- 32.5 Lorentzian Lattices . . . 86
- 32.6 Bibliography . . . 87

33 Lattices with Group Action **89**
- 33.1 Introduction . . . 89
- 33.2 Automorphism Group and Isometry Testing . . . 89
- 33.3 Lattices from Matrix Groups . . . 95
- 33.4 Bibliography . . . 105

34 Quadratic Forms **107**
- 34.1 Introduction . . . 107
- 34.2 Constructions and Conversions . . . 107
- 34.3 Local Invariants . . . 107
- 34.4 Isotropic Subspaces . . . 109
- 34.5 Equivalence . . . 111
- 34.6 Bibliography . . . 112

35 Binary Quadratic Forms **113**
- 35.1 Introduction . . . 113
- 35.2 Creation Functions . . . 113
- 35.3 Basic Invariants . . . 114
- 35.4 Operations on Forms . . . 114
- 35.5 Reduced Forms . . . 117
- 35.6 Class Groups . . . 117

35.7 Discrete Logarithms . . . 119
35.8 Elliptic and Modular Invariants . . . 119
35.9 Class Invariants . . . 120
35.10 Bibliography . . . 120

References **121**

Index **124**

Chapter 31

Lattices

31.1 Introduction

Lattices play an important role in various areas, e.g., representation theory, coding theory, geometry, and algebraic number theory. A lattice in Magma is a free $\mathbb{Z}$-module contained in $\mathbb{Q}^n$ or $\mathbb{R}^n$, together with a positive definite inner product having image in $\mathbb{Q}$ or $\mathbb{R}$. The information specifying a lattice is a basis, given by a sequence of elements in $\mathbb{Q}^n$ or $\mathbb{R}^n$, and a bilinear product $(\cdot, \cdot)$, given by $(v, w) = vMw^{tr}$ for a positive definite matrix M. Central to the lattice machinery in Magma is a fast implementation of the *Lenstra-Lenstra-Lovász lattice reduction* algorithm [LLL82]. The Lenstra-Lenstra-Lovász algorithm (LLL for short) takes a basis of a lattice and returns a new basis of the lattice which is *LLL-reduced* which usually means that the vectors of the new basis have small norms. The Magma algorithm is based on both the floating-point LLL algorithm of Nguyen and Stehlé [NS09a] and the exact integral algorithm as described by de Weger in [dW87], but includes many optimisations, with particular attention to different kinds of input matrices. The algorithm can operate on either a basis matrix or a Gram matrix and can be controlled by many parameters (selection of methods, LLL reduction constants, step and time limits, etc.). For each lattice, a LLL-reduced basis for the lattice is computed and stored internally when necessary and subsequently used for many operations. This gives maximum efficiency for the operations, yet all the operations are presented using the external ("user") basis of the lattice. Lattices are thus a more efficient alternative to R-spaces where R is the integer ring $\mathbb{Z}$ since lattices use a LLL-reduced form of the basis matrix extensively instead of the Hermite form of a basis matrix used by R-spaces which may have very large entries. Another important component of the lattice machinery is a very efficient algorithm for enumerating all vectors of a lattice with norms in a given range. This algorithm is used for computing the minimum, the shortest vectors, vectors up to a chosen length, and vectors close to or closest to a given vector (possibly) outside a lattice. Several interesting lattices are directly accessible inside Magma using standard constructions (e.g., root lattices and preimages of linear codes). Additionally, an interface has been provided to convert lattices in the database of G. Nebe and N.J.A. Sloane [NS01a] into Magma format.

31.2 Presentation of Lattices

A lattice in Magma is a free $\mathbb{Z}$-module of rank m inside $\mathbb{Z}^n$, $\mathbb{Q}^n$, or $\mathbb{R}^n$, where $m \leq n$. We call m the rank or dimension of the lattice and n its degree. A lattice with $m = n$ is called a *full* lattice. There are two pieces of information which completely determine a lattice L: the basis matrix B and the inner product matrix M. M is always a positive definite matrix (i.e., a matrix M such that $vMv^{tr} > 0$ for all non-zero vectors $v \in \mathbb{R}^n$). All other information associated with a lattice L is derived from B and M. For example, the inner product $(\cdot, \cdot)$ for L is given by $(v, w) = vMw^{tr}$ and thus the Gram matrix F for L giving the inner products of the basis vectors is BMB^{tr}. Lattices differ from R-spaces in that a lattice is always a $\mathbb{Z}$-module even if entries of some elements of the lattice are not integers. For R-spaces the base ring R affects the module structure but for lattices the "base ring" R is just defined to be the smallest ring over which the basis and inner product matrices can be represented. Magma basically has two kinds of lattices: *exact* lattices for which all of the entries of B and M (and thus F) lie in either $\mathbb{Z}$ or $\mathbb{Q}$, and *non-exact* lattices for which all of the entries of B and M (and thus F) lie in a particular

approximate real field $\mathbb{R}$. For exact lattices, the "base ring" R may be $\mathbb{Z}$ or $\mathbb{Q}$ as mentioned above so that elements are represented over R, but otherwise there is nothing in the lattice which distinguishes between the $\mathbb{Z}$ and $\mathbb{Q}$ cases. When exact lattices are printed, the basis matrix B is presented uniquely as an integral matrix divided by a minimal positive integral denominator. The inner product matrix M is presented similarly. This tends to be a more natural presentation which reflects the $\mathbb{Z}$-module structure. Another useful feature of this presentation is that the inner product can be efficiently defined to be the identity matrix divided by an appropriate integral scale factor. In this way, the basis can be kept rational while still obtaining an integral primitive Gram matrix (a matrix with integral entries which are coprime), thus avoiding the need for rescaling the basis by an irrational square root. This method is followed in the construction of special lattices provided within Magma. For non-exact lattices, the basis matrix B is simply presented as it is represented over the ring R which is an approximate real field. Some operations applicable for exact lattices are not possible for non-exact lattices. Thus if in the description of a function below it is said that the function can only be applied to an exact lattice it means that the function cannot be applied to a lattice over an approximate real field. Two lattices are said to be *compatible* if their inner product matrices are equal. Many operations on lattices assume that their arguments are compatible. Intuitively, two lattices and their elements are not comparable if their inner product matrices are different. Associated with a lattice L is its coordinate lattice L' which has the standard basis (i.e., its basis matrix is the identity matrix) but has the same Gram matrix as L. Some operations, e.g., automorphism group, always work with the coordinates of elements of the lattice, so it is necessary to move to the coordinate lattice. The category of lattices is `Lat`.

31.3 Creation of Lattices

Lattices can be created in elementary ways whereby the basis matrix or a generating matrix and possibly also the inner product matrix are specified. Magma also provides functions for creating lattices from linear codes or algebraic number fields and for creating some special lattices.

31.3.1 Elementary Creation of Lattices

This subsection describes the elementary ways of creating lattices by supplying the basis or the Gram matrix.

There are two ways of specifying the basis of a lattice at creation. First, a generating matrix can be specified whose rows need not be independent; the matrix is reduced to a LLL-reduced form and zero rows are removed to yield a basis matrix. Secondly, a basis matrix (with independent rows) can be specified; this matrix is used for the basis matrix without any changes being made to it. As well as the basis, a particular inner product can also be specified by a symmetric positive definite matrix. By default (when a particular inner product matrix is not given), the inner product is taken to be the standard Euclidean product. Instead of giving the basis one can also define a lattice by its Gram matrix F which prescribes the inner products of the basis vectors. The basis is taken to be the standard basis and the inner product matrix is taken as F so that the Gram matrix for the lattice is also F.

```
Lattice(X, M): ModMatRngElt, AlgMatElt -> Lat
Lattice(n, Q, M): RngIntElt, [ RngElt ], AlgMatElt -> Lat
Lattice(S, M): ModTupRng, AlgMatElt -> Lat
```

```
CheckPositive: BoolElt                    Default: true
```

Construct the lattice L specified by the given generating matrix and with inner product given by the $n \times n$ matrix M. The generating matrix can be specified by an $r \times n$ matrix X, an integer n with a sequence Q of ring elements of length rn (interpreted as a $r \times n$ matrix), or an R-space S of dimension r and degree n. In each case, the generating matrix need not have independent rows (though it always will in the R-space case). The matrix is reduced to a LLL-reduced form and zero rows are removed to yield a basis matrix B of rank m (so $m \leq r$). The lattice L is then constructed to have rank m, degree n, basis matrix B, and inner product matrix M. By default Magma checks that M is positive definite but by setting `CheckPositive := false` this check will be suppressed. (Unpredictable behaviour will follow if unchecked incorrect input is given.)

```
Lattice(X): ModMatRngElt -> Lat
```

`Lattice(n, Q): RngIntElt, [ RngElt ] -> Lat`

`Lattice(S): ModTupRng -> Lat`

Construct the lattice L specified by the given generating matrix and with standard Euclidean inner product. The generating matrix can be specified by an $r \times n$ matrix X, an integer n with a sequence Q of ring elements of length rn (interpreted as a $r \times n$ matrix), or an R-space S of dimension r and degree n. In each case, the generating matrix need not have independent rows (though it always will in the R-space case). The matrix is reduced to a LLL-reduced form and zero rows are removed to yield a basis matrix B of rank m (so $m \leq r$). The lattice L is then constructed to have rank m, degree n, basis matrix B, and standard Euclidean inner product.

`LatticeWithBasis(B, M): ModMatRngElt, AlgMatElt -> Lat`

`LatticeWithBasis(n, Q, M): RngIntElt, [ RngElt ], AlgMatElt -> Lat`

`LatticeWithBasis(S, M): ModTupRng, AlgMatElt -> Lat`

```
CheckIndependent: BoolElt                    Default: true
CheckPositive   : BoolElt                    Default: true
```

Construct the lattice L with the specified $m \times n$ basis matrix and with inner product given by the $n \times n$ matrix M. The basis matrix B can be specified by an $m \times n$ matrix B, an integer n with a sequence Q of ring elements of length mn (interpreted as an $m \times n$ matrix B), or an R-space S of dimension m and degree n (whose basis matrix is taken to be B). The rows of B must be independent (i.e., form a basis). The lattice L is then constructed to have rank m, degree n, basis matrix B, and inner product matrix M. (Note that the basis matrix B is *not* reduced to a LLL-reduced form as in the `Lattice` function.) By default Magma checks that the rows of the matrix B specifying the basis are independent but by setting `CheckIndependent := false` this check will be suppressed. By default Magma also checks that M is positive definite but by setting `CheckPositive := false` this check will be suppressed. (Unpredictable behaviour will follow if unchecked incorrect input is given.)

`LatticeWithBasis(B): ModMatRngElt -> Lat`

`LatticeWithBasis(n, Q): RngIntElt, [ RngElt ] -> Lat`

`LatticeWithBasis(S): ModTupRng -> Lat`

```
CheckIndependent: BoolElt                    Default: true
```

Construct the lattice L with the specified $m \times n$ basis matrix and with standard Euclidean inner product. The basis matrix B can be specified by an $m \times n$ matrix B, an integer n with a sequence Q of ring elements of length mn (interpreted as an $m \times n$ matrix B), or an R-space S of dimension m and degree n (whose basis matrix is taken to be B). The rows of B must be independent (i.e., form a basis). The lattice L is then constructed to have rank m, degree n, basis matrix B, and standard Euclidean product. (Note that the basis matrix B is *not* reduced to a LLL-reduced form as in the `Lattice` function.) By default Magma checks that the rows of the matrix B specifying the basis are independent but by setting `CheckIndependent := false` this check will be suppressed. (Unpredictable behaviour will follow if unchecked incorrect input is given.)

`LatticeWithGram(F): AlgMatElt -> Lat`

`LatticeWithGram(n, Q): RngIntElt, [RngElt] -> Lat`

```
CheckPositive: BoolElt                    Default: true
```

Construct a lattice with the given Gram matrix. The Gram matrix F can be specified as a symmetric $n \times n$ matrix F, or an integer n and a sequence Q of ring elements of length n^2 or $\binom{n+1}{2}$. In the latter case, a sequence of length n^2 is regarded as an $n \times n$ matrix and a sequence of length $\binom{n+1}{2}$ as a lower triangular matrix determining a symmetric matrix F.

This function constructs the lattice L of degree n having the standard basis and inner product matrix F, therefore having Gram matrix F as well. By default Magma checks that F is positive definite but by setting `CheckPositive := false` this check will be suppressed. (Unpredictable behaviour will follow if unchecked incorrect input is given.)

`StandardLattice(n): RngIntElt -> Lat`

Create the standard lattice $\mathbb{Z}^n$ with standard Euclidean inner product.

`CoordinateLattice(L): Lat -> Lat`

Constructs the lattice with the same Gram matrix as the lattice L, but with basis equal to the canonical basis of the free module of $\mathrm{Rank}(L)$ over the base ring of L. The identification of basis elements thus determines an isometry of lattices.

`Module(L): Lat -> Lat`

`ZLattice(L): Lat -> Lat`

Returns the lattice L.

`ScaledLattice(L, n): Lat, RngIntElt -> Lat`

`ScaledLattice(L, n): Lat, FldRatElt -> Lat`

Constructs the coordinate lattice with Gram matrix equal to that of the lattice L scaled by the integer or rational n. The identification of basis elements thus determines an similitude of lattices.

`TernaryQuadraticLattice(d): RngIntElt -> Lat`

For a cube-free integer d, Constructs a positive definite ternary quadratic lattice of half-discriminant d.

`QuaternaryLatticeOfPrimeDiscriminant(p): RngIntElt -> Lat`

Constructs a positive-definite quaternary quadratic lattice of prime discriminant p.

`QuinaryLatticeOfPrimeDiscriminant(p): RngIntElt -> Lat`

Constructs a positive-definite quinary quadratic lattice of prime half-discriminant p.

`QuadraticLattice(n, d): RngIntElt, RngIntElt -> Lat`

Given a positive integer $n \leq 6$ and a discriminant d, constructs a positive-definite quadratic lattice of dimension n and discriminant d using Minkowski's description of the space of reduced forms.

`RandomLattice(n, M): RngIntElt, RngIntElt -> Lat`

```
Positive: BoolElt                    Default: false
```

Generates a random lattice. If `Positive` is `true`, generates a positive definite lattice.

Example: Creating a Lattice in Four Ways (ex-d7d03e)

We create the lattice in $\mathbb{Z}^3$ of degree 3 and rank 2 generated by the vectors $(1, 2, 3)$ and $(3, 2, 1)$ in four different ways. The first two (identical) lattices are represented by LLL-reduced bases since the `Lattice` function is used to create them, while the latter two (identical) lattices remain represented by the original basis since the `LatticeWithBasis` function is used to create them.

```
> B := RMatrixSpace(IntegerRing(), 2, 3) ! [1,2,3, 3,2,1];
> B;
[1 2 3]
[3 2 1]
> L1 := Lattice(B);
> L1;
Lattice of rank 2 and degree 3
Basis:
( 2  0 -2)
( 1  2  3)
> L2 := Lattice(3, [1,2,3, 3,2,1]);
> L2 eq L1;
true
> L3 := LatticeWithBasis(B);
> L3;
Lattice of rank 2 and degree 3
```

(continues on next page)

(continued from previous page)

```
Basis:
(1 2 3)
(3 2 1)
> L4 := LatticeWithBasis(3, [1,2,3, 3,2,1]);
> L4 eq L3, L1 eq L3;
true true
```

We next create a 3×3 positive definite matrix M and create the lattice L with basis the same as above but also with inner product matrix M. We note the Gram matrix of the lattice which equals BMB^{tr} where B is the basis matrix.

```
> B := RMatrixSpace(IntegerRing(), 2, 3) ! [1,2,3, 3,2,1];
> M := MatrixRing(IntegerRing(), 3) ! [3,-1,1, -1,3,1, 1,1,3];
> M;
[ 3 -1  1]
[-1  3  1]
[ 1  1  3]
> IsPositiveDefinite(M);
true
> L := LatticeWithBasis(B, M);
> L;
Lattice of rank 2 and degree 3
Basis:
(1 2 3)
(3 2 1)
Inner Product Matrix:
[ 3 -1  1]
[-1  3  1]
[ 1  1  3]
> GramMatrix(L);
[56 40]
[40 40]
```

Finally, we create a lattice C with the same Gram matrix as the last lattice, but with standard basis. To do this we use the `LatticeWithGram` function. This lattice C is the *coordinate* lattice of L: it has the same Gram matrix as L but is not compatible with L since its inner product matrix is different to that of L.

```
> F := MatrixRing(IntegerRing(), 2) ! [56,40, 40,40];
> C := LatticeWithGram(F);
> C;
Standard Lattice of rank 2 and degree 2
Inner Product Matrix:
[56 40]
[40 40]
> GramMatrix(C);
[56 40]
[40 40]
> C eq CoordinateLattice(L);
true
> GramMatrix(C) eq GramMatrix(L);
true
> C eq L;

Runtime error in 'eq': Arguments are not compatible
```

31.3.2 Lattices from Linear Codes

This subsection presents some standard constructions (known as constructions 'A' and 'B') to obtain lattices from linear codes. In some cases the structural invariants of these lattices (e.g., minimum and kissing number, hence the centre density) can be deduced from invariants of the codes; in general one still gets estimates for the invariants of the lattices.

`Lattice(C, "A"): Code -> Lat`

Let C be a linear code of length n over a prime field $K := \mathbf{F}_p$. Construct the lattice $L \subseteq \mathbb{Z}^n$ which is the full preimage of C under the natural epimorphism from $\mathbb{Z}^n$ onto K^n.

`Lattice(C, "B"): Code -> Lat`

Let C be a linear code of length n over a prime field $K := \mathbf{F}_p$ such that all code words have coordinate sum 0. Construct the lattice $L \subseteq \mathbb{Z}^n$ which consists of all vectors reducing modulo p to a code word in C and whose coordinate sum is 0 modulo p^2. The inner product matrix is set to the appropriate scalar matrix so that the Gram matrix is integral and primitive (its entries are coprime).

Example: Barnes-Wall Lattice from a Reed-Muller Code `(ex-60a64b)`

The 16-dimensional Barnes-Wall lattice Λ_{16} can be constructed from the first order Reed-Muller code of length 16 using construction 'B'. Note that the inner product matrix is the identity matrix divided by 2 so that the Gram matrix is integral and primitive.

```
> C := ReedMullerCode(1, 4);
> C: Minimal;
[16, 5, 8] Reed-Muller Code (r = 1, m = 4) over GF(2)
> L := Lattice(C, "B");
> L;
Lattice of rank 16 and degree 16
Basis:
(1 0 0 1 0 1 1 0 0 1 1 0 1 0 0 1)
(0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1)
(0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1)
(0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 2)
(0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1)
(0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 2)
(0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 2)
(0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 2)
(0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1)
(0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 2)
(0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 2)
(0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 2)
(0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 2)
(0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 2)
(0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 2)
(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4)
Inner Product denominator: 2
```

31.3.3 Lattices from Algebraic Number Fields

Let K be an algebraic number field of degree n over $\mathbb{Q}$. Then K has n embeddings into the complex numbers, denoted by $\sigma_1, \ldots, \sigma_n$. By convention the first s of these are embeddings into the real numbers and the last $2t$ are pairs of complex embeddings such that $\sigma_{s+t+i} = \overline{\sigma_{s+i}}$ for $1 \leq i \leq t$. The T_2-norm $K \to \mathbb{R} : \alpha \mapsto \sum_{i=1}^{n} |\sigma_i(\alpha)|^2$ defines a positive definite norm on K in which an order or ideal is a positive definite lattice. We restrict to the r real embeddings and the first s complex embeddings to obtain an embedding of a rank n order or ideal $\mathbb{R}^s \times \mathbb{C}^s = \mathbb{R}^n$. By rescaling the complex coordinates by $\sqrt{2}$, we recover the T_2-norm of the element as its Euclidean norm under the embedding in $\mathbb{R}^n$. This embedding,

given by

$$\alpha \mapsto \left(\sigma_1(\alpha), \ldots, \sigma_s(\alpha), \sqrt{2}\,\mathrm{Re}(\sigma_{s+1}(\alpha)), \sqrt{2}\,\mathrm{Im}(\sigma_{s+1}(\alpha)), \ldots, \sqrt{2}\,\mathrm{Im}(\sigma_{s+t}(\alpha))\right),$$

is called the *Minkowski map* on K.

MinkowskiLattice(O): RngOrd -> Lat, Map

Lattice(O): RngOrd -> Lat, Map

Given an order O in a number field of degree n, create the lattice L in $\mathbb{R}^n$ generated by the images of the basis of O under the Minkowski map, followed by the isomorphism $O \to L$.

MinkowskiLattice(I): RngOrdIdl -> Lat, Map

Lattice(I): RngOrdIdl -> Lat, Map

Given an ideal I of an order O in a number field of degree n, create the lattice in $\mathbb{R}^n$ generated by the images of the basis of I under the Minkowski map, followed by the isomorphism $O \to L$.

MinkowskiSpace(K): FldNum -> KModTup, Map

Construct the real inner product space V defined with respect to the inner product matrix of the T_2-norm on the number field K, followed by the Minkowski map $K \to V$.

Example: Lattice of an Equation Order (ex-12a121)

In this example we create the lattice corresponding to the equation order of the number field $\mathbb{Q}(\sqrt[3]{15})$. For comparison we apply the Minkowski map to the elements of the basis for the order.

```
> P<x> := PolynomialRing(Integers());
> R := EquationOrder(NumberField(x^3-15));
> w := R![0,1,0];
> L, f := MinkowskiLattice(R);
> L;
Lattice of rank 3 and degree 3 over Real Field
Basis:
Lattice of rank 3 and degree 3 over Real field of precision 30
Basis:
(1.0000000000000000000000000000 1.4142135623730950488016887242l
    0.0000000000000000000000000000)
(2.4662120743304701014916113231 5 -1.7438752816032172007962969421 6
    3.0204805898002556568833586998 8)
(6.0822019955734001848984449977 4 -4.3007662756163029791484373981 2
    -7.4491457008462102763572729513 9)
> B := Basis(R);
> f(B[2]);
> f(B[2]);
(2.4662120743304701014916113231 5 -1.7438752816032172007962969421 6
    3.0204805898002556568833586998 8)
```

Similarly, we create the lattice defined by the ideal generated by $w^2 + 1$, and verify the that the T_2-norm, given by **Length** on the number field, agrees with the norm of the lattice image.

```
> I := ideal< R | w^2+1 >;
> L, f := Lattice(I);
> B := Basis(I);
> [ Length(B[k]) : k in [1..3] ];
[ 113.97954334487115456712250954195650692309680189826241920804543
33463201962881530385844733441619895572252810601641408175119302890
04933259997863098975005711709389244772372 4,
693.24660598672020055469533499321340725963301213152993696031825 5
63453137091693843872540429293365290275285892767413846480567664278
88120209935022451021109703715590090765 24,
```

(continues on next page)

(continued from previous page)

```
4216.46589035691627937357288301497314034052453149249823527965295
12327604191926790990606993516917986916460693283225627563965475 17
8203732351663613772502545069466948994156 ]
> [ Norm(f(B[k])) : k in [1..3] ];
[ 113.97954334487115456712250954 2,
693.24660598672020055469533499 3, 4216.46589035691627937357288302
]
```

31.3.4 Special Lattices

This subsection presents functions to construct some special lattices, namely root lattices, laminated lattices, and Kappa-lattices. A much wider variety of lattices can be found in a database provided by G. Nebe and N.J.A. Sloane at the web-site [NS01a]. These lattices can easily be made accessible to Magma by a conversion script also available at this web-site. The database currently contains (at least):

- The root lattices A_n and their duals for $1 \leq n \leq 24$
- The root lattices D_n and their duals for $1 \leq n \leq 24$
- The root lattices E_n and their duals for $6 \leq n \leq 8$
- The laminated lattices Λ_n for $1 \leq n \leq 24$ including the 16-dimensional Barnes-Wall lattice Λ_{16} and the Leech lattice Λ_{24}
- The Kappa-lattices K_n and their duals for $7 \leq n \leq 13$ including the Coxeter-Todd lattice K_{12}
- The perfect lattices up to dimension 7
- The 3-dimensional Bravais lattices
- Various interesting lattices in dimensions 20, 24, 28, 32, 40, 80, 105 including, e.g., some of the densest known lattices in dimension 32.

`Lattice(X, n): MonStgElt, RngIntElt -> Lat`

Given a family name X as a string which is one of `"A"`, `"B"`, `"C"`, `"D"`, `"E"`, `"F"`, `"G"`, `"Kappa"`, or `"Lambda"`, together with an integer n, construct a lattice subject to the following specifications:

`A`:
The root lattice A_n which is the zero-sum lattice in $\mathbb{Q}^{n+1}$.

`B`:
$n \geq 2$: The root lattice B_n which is the standard lattice of dimension n.

`C`:
$n \geq 3$: The root lattice C_n which is the even sublattice of $\mathbb{Z}^n$ and is equal to D_n.

`D`:
$n \geq 3$: The root lattice D_n which is the even sublattice of $\mathbb{Z}^n$, also called the checkerboard lattice.

`E`:
$6 \leq n \leq 8$: The root lattice E_n, also called Gosset lattice.

`F`:
$n = 4$: The root lattice F_4 which is equal to D_4.

`G`:
$n = 2$: The root lattice G_2 which is equal to A_2.

`Kappa`:
$1 \leq n \leq 13$: The Kappa-lattice K_n. For $n = 12$ this is the Coxeter-Todd lattice.

`Lambda`:
$1 \leq n \leq 31$: The laminated lattice Λ_n. For $n = 16$ this is the Barnes-Wall lattice, for $n = 24$ the Leech lattice. To avoid irrational entries, each lattice is presented with inner product

matrix taken to be the identity matrix divided by a suitable scale factor so that the Gram matrix is integral and primitive (its entries are coprime).

31.4 Lattice Elements

The following functions allow basic operations on elements of lattices. The elements of lattices are simply (row) vectors, just as for R-spaces. Most of the operations for R-space elements are also applicable to lattice elements.

31.4.1 Creation of Lattice Elements

`L . i: Lat, RngIntElt -> LatElt`

Return the i-th basis element of the current basis of the lattice L.

`L ! Q: Lat, [ RngElt ] -> LatElt`

`elt<L | Q>: Lat, [ RngElt ] -> LatElt`

Given a lattice L of degree n and a sequence Q of length n, create the lattice element with the corresponding sequence elements as entries. The sequence must consist of elements coercible into the base ring of L. The resulting vector must lie within L.

`CoordinatesToElement(L, C): Lat, [ RngIntElt ] -> LatElt`

`Coordelt(L, C): Lat, [ RngIntElt ] -> LatElt`

`CoordinatesToElement(L, C): Lat, Mtrx -> LatElt`

`Coordelt(L, C): Lat, Mtrx -> LatElt`

Given a lattice L of rank m and a sequence or vector $C = [c_1, \ldots, c_m]$ of length m of *integers*, create the lattice element $c_1 \cdot b_1 + \ldots + c_m \cdot b_m$, where $[b_1, \ldots, b_m]$ is the basis of L.

`L ! 0: Lat, RngIntElt -> LatElt`

`Zero(L): Lat -> LatElt`

Return the zero element of the lattice L.

31.4.2 Operations on Lattice Elements

`- v: LatElt -> LatElt`

Given an element v in a lattice L, return its negation $-v$ in L.

`v + w: LatElt, LatElt -> LatElt`

Given elements v and w in a lattice L, return the sum $v + w$ in L.

`v - w: LatElt, LatElt -> LatElt`

Given elements v and w in a lattice L, return the difference $v - w$ in L.

`v * s: LatElt, RngIntElt -> .`

`s * v: RngIntElt, LatElt -> .`

Given an element v in a lattice L and a scalar s of the ring S, return the product $s \cdot v$ (scalar multiplication of v by s). If s is an integer, the resulting vector will lie in L; otherwise the resulting vector will lie in the R-space of the appropriate degree whose coefficient ring is the parent S of s.

`v / s: LatElt, RngIntElt -> .`

Given an element v in a lattice L and a scalar s of the ring S, return the product $(1/s) \cdot v$ (scalar multiplication of v by $1/s$). The resulting vector will always lie in the R-space of the appropriate degree whose coefficient ring is the field of fractions of the parent S of s.

`v div d: LatElt, RngIntElt -> LatElt`

Given an element v in a lattice L and an *integer* d, return the vector $(1/d) \cdot v$ (scalar multiplication of v by $1/d$) as an element of L if the scaled vector lies in L. If the scaled vector does not lie in L, an error ensues. Note that this is different from v/d.

`v +:= w: LatElt, LatElt`

(Assignment statement.) Replace lattice element v by the sum $v + w$.

`v -:= w: LatElt, LatElt`

(Assignment statement.) Replace lattice element v by the difference $v - w$.

`v *:= n: LatElt, RngIntElt`

(Assignment statement.) Replace lattice element v by the scalar product $n \cdot v$, where n is an integer.

`v * T: LatElt, AlgMatElt -> LatElt`

`v * T: LatElt, ModMatRngElt -> LatElt`

`v * T: LatElt, GrpMatElt -> LatElt`

Given an element v in a lattice L of degree n, return the result of multiplying v from the right by the $n \times n$ matrix T. The matrix T may be any matrix which is n by n and over the base ring of L. The resulting product must lie in the lattice L.

`InnerProduct(v, w): LatElt, LatElt -> RngElt`

`(v, w)`

Given elements v and w of a lattice L, return their inner product (v, w) with respect to the inner product of L. This is vMw^{tr} where M is the inner product matrix of L.

`Norm(v): LatElt -> RngElt`

Given an element v of a lattice L, return its norm (v, v) with respect to the inner product of L. This is vMv^{tr} where M is the inner product matrix of L. Note that in the case of a lattice with standard Euclidean inner product this is the square of the usual Euclidean length.

`Length(v, K): LatElt, Fld -> FldReElt`

`Length(v): LatElt -> FldReElt`

Given an element v of a lattice L, return its length $\sqrt{(v, v)}$ with respect to the inner product of L as an element of the real field K. This is $\sqrt{vMv^{tr}}$ where M is the inner product matrix of L. The argument for the real field K may be omitted, in which case K is taken to be the current default real field. In the case of a lattice with standard Euclidean inner product this is the usual Euclidean length.

`Support(v): LatElt -> SetEnum`

Given an element v of a lattice L, return its support, i.e., the numbers of the columns at which v has non-zero entries.

31.4.3 Predicates and Boolean Operations

`v in L: LatElt, Lat -> BoolElt`

Given an element v of a lattice which is compatible with the lattice L, return `true` if and only if v is in L.

`v eq w: LatElt, LatElt -> BoolElt`

Given elements v and w of a lattice L, return `true` if and only if lattice elements v and w of lattice L are equal.

`v ne w: LatElt, LatElt -> BoolElt`

Given elements v and w of a lattice L, return `false` if and only if lattice elements v and w of lattice L are equal.

`IsZero(v): LatElt -> BoolElt`

Given an element v of a lattice L, return `true` if and only if v is the zero element L.

31.4.4 Access Operations

`ElementToSequence(v): LatElt -> [ RngElt ]`

`Eltseq(v): LatElt -> [ RngElt ]`

Given an element v of a lattice L of degree n, return the sequence of entries of v of length n.

`Coordinates(v): LatElt -> [ RngIntElt ]`

Given an element v of a lattice L having $\mathbb{Z}$-basis $[b_1, \ldots, b_m]$, return a sequence $[c_1, \ldots, c_m]$ of elements of $\mathbb{Z}$ giving the (unique) coordinates of v relative to the $\mathbb{Z}$-basis, so that $v = c_1 \cdot b_1 + \ldots + c_m \cdot b_m$.

`Coordinates(L, v): Lat, LatElt -> [ RngIntElt ]`

Given a lattice L of degree n having $\mathbb{Z}$-basis $[b_1, \ldots, b_m]$, together with an element v of a lattice L' (also of degree n), return a sequence $[c_1, \ldots, c_m]$ of elements of $\mathbb{Z}$ giving the (unique) coordinates of v relative to the $\mathbb{Z}$-basis of L, so that $v = c_1 \cdot b_1 + \ldots + c_m \cdot b_m$.

`CoordinateVector(v): LatElt -> LatElt`

Given an element v of a lattice L having $\mathbb{Z}$-basis $[b_1, \ldots, b_m]$, return the vector $c = (c_1, \ldots, c_m)$ of the coordinate lattice C of L (see the function `CoordinateLattice`) giving the (unique) coordinates of v relative to the $\mathbb{Z}$-basis, so that $v = c_1 \cdot b_1 + \ldots + c_m \cdot b_m$.

`CoordinateVector(L, v): Lat, LatElt -> LatElt`

Given a lattice L of degree n having $\mathbb{Z}$-basis $[b_1, \ldots, b_m]$, together with an element v of a lattice L' (also of degree n), return the vector $c = (c_1, \ldots, c_m)$ of the coordinate lattice C of L (see the function `CoordinateLattice`) giving the (unique) coordinates of v relative to the $\mathbb{Z}$-basis of L, so that $v = c_1 \cdot b_1 + \ldots + c_m \cdot b_m$.

Example: Operations on Lattice Elements (ex-0121de)

This example demonstrates simple uses of the operations on lattice elements.

```
> L := LatticeWithBasis(3, [1,0,0, 1,2,3, 3,6,2]);
> L;
Lattice of rank 3 and degree 3
Basis:
(1 0 0)
(1 2 3)
(3 6 2)
> Coordelt(L, [1, 2, 1]);
( 6 10  8)
> v := L.2;
> w := L ! [2, 4, 6];
> Eltseq(v);
[ 1, 2, 3 ]
> Coordinates(w);
[ 0, 2, 0 ]
> Coordelt(L, [1, 1, 1]);
(5 8 5)
> Norm(v);
14
> InnerProduct(v, w);
28
> A := MatrixRing(Integers(), 3);
> X := A ! [0,-1,0, 1,0,0, 0,1,2];
> X;
[ 0 -1  0]
[ 1  0  0]
[ 0  1  2]
> u := L.1 + L.3;
```

(continues on next page)

(continued from previous page)

```
> Determinant(X);
2
> Norm(u);
56
> u * X;
( 6 -2  4)
> Norm(u * X);
56
```

31.5 Properties of Lattices

The following functions provide access to the elementary attributes and properties of lattices. Other attributes and invariants of a lattice, e.g., successive minimum, kissing number, and theta series are described in subsequent sections.

31.5.1 Associated Structures

AmbientSpace(L): Lat -> ModTupFld, Map

The ambient rational or real vector space in which the lattice L embeds, followed by the embedding map.

CoordinateSpace(L): Lat -> ModTupFld, Map

The ambient vector space of the coordinate lattice, i.e., the vector space of dimension equal to the rank of the lattice L, with inner product matrix equal to the Gram matrix of L. The embedding map is returned as the second return value.

Category(L): Lat -> Cat

Type(L): Lat -> Cat

Returns the category Lat of lattices.

31.5.2 Attributes of Lattices

Dimension(L): Lat -> RngIntElt

Rank(L): Lat -> RngIntElt

Return the rank of the lattice L, which equals the number of basis elements in L. Note that the rank of the lattice may be smaller than its degree n, which is the dimension of the real space $\mathbb{R}^n$ in which L is defined.

Degree(L): Lat -> RngIntElt

Return the degree of the lattice L, which is the dimension n of the real space $\mathbb{R}^n$ in which L is defined.

Degree(v): LatElt -> RngIntElt

Return the degree of the lattice element v, which the degree of the lattice to which it belongs.

Content(L): Lat -> RngElt

Given an exact lattice L, return the largest rational number c such that $(u, v) \in c\mathbb{Z}$ for all $u, v \in L$.

Level(L): Lat -> RngElt

Given an integral lattice L, return the smallest integer k such that $k(v, v) \in 2\mathbb{Z}$ for all v in the dual of L.

Determinant(L): Lat -> RngElt

Returns the determinant of the lattice L, which is defined to be the determinant of the Gram matrix F of L. For a full rank lattice the square root of Determinant(L) is the volume of a fundamental parallelotope of the lattice.

`Discriminant(L): Lat -> RngInt`

Returns the discriminant of the lattice L, which is defined to be the ideal generated by its determinant. When L is of odd rank, returns half of this quantity, the half-discriminant.

`Scale(L): Lat -> RngIntFracIdl`

The scale of the lattice L, which is the ideal generated by the inner products of pairs of lattice elements. This is the ideal generated by the content of L.

`BadPrimes(L): Lat -> Set`

The set of all prime ideals at which L is not unimodular.

`GramMatrix(L): Lat -> AlgMatElt`

`AuxiliaryForms(L): Lat -> [ AlgMatElt ]`

Return the Gram matrix for the lattice L of rank m, which is the $m \times m$ matrix $F = BMB^{tr}$, where B is the basis matrix of L and M is the inner product matrix of L. Thus the (i, j)-th entry of F equals the inner product of the basis vectors b_i and b_j of L.

`GramMatrix(L, S): Lat, [ ModTupFldElt ] -> AlgMatElt`

`GramMatrix(S): [ LatElt ] -> AlgMatElt`

`Half: BoolElt` Default: `false`

Given a lattice L and a sequence S of vectors in the lattice, return the Gram matrix of these vectors. If `Half` is set to `true`, scales the Gram matrix by a half.

`GramMatrix(X): ModMatRngElt -> AlgMatElt`

Given a matrix X, return XX^{tr}. Note that this function will take half the time as would be taken for the invocation `X*Transpose(X)` since the symmetry of the result is taken advantage of.

`InnerProductMatrix(L): Lat -> AlgMatElt`

The inner product matrix M of the lattice L, which is an $n \times n$ matrix, where n is the degree of L. If L has the standard Euclidean product, M is the identity matrix.

`Basis(L): Lat -> [ FldReElt ]`

`Basis(L): Lat -> [ FldRatElt ]`

`Basis(L): Lat -> [ RngIntElt ]`

`PseudoBasis(L): Lat -> [ RngIntElt ]`

Return the basis of the lattice L as a sequence $[b_1, \ldots, b_m]$ of elements of L.

`BasisMatrix(L): Lat -> ModMatRngElt`

Return the $m \times n$ matrix having the basis elements of L as rows, where m is the rank and n the degree of L. The coefficient ring of the matrix is the same as the base ring of L.

`BasisDenominator(L): Lat -> RngIntElt`

Given an exact lattice L, return the common denominator of the entries of the current basis of L.

`CoefficientIdeals(L): Lat -> [ RngInt ]`

Given a lattice L of rank n, return its coefficient ideals, which are n copies of the trivial ideal.

`Involution(L): Lat -> FldAut`

Given a lattice L, return the involution with respect to which it is sesquilinear, which is the identity.

`QuadraticForm(L): Lat -> RngMPolElt`

The quadratic form of the lattice L as a multivariate polynomial.

31.5.3 Predicates and Booleans on Lattices

`L eq M: Lat, Lat -> BoolElt`

Given lattices L and M, return `true` if and only if the lattices are subsets of one another.

`L ne M: Lat, Lat -> BoolElt`

The logical negation of `eq`.

`L subset M: Lat, Lat -> BoolElt`

Return `true` if and only if L is a sublattice of M, i.e., both L and M are lattices, and L is a subset of M.

`IsZero(L): Lat -> BoolElt`

Checks if the rank of L is 0.

`IsFull(L): Lat -> BoolElt`

Return `true` if L has full rank.

`IsHermitian(L): Lat -> BoolElt`

Return `false`.

`IsQuadratic(L): Lat -> BoolElt`

Return `true` if and only if L is a quadratic lattice.

`IsExact(L): Lat -> BoolElt`

Return `true` if and only if L is an exact lattice, i.e., the coefficient ring of L is $\mathbb{Z}$ or $\mathbb{Q}$ and $(v, w) \in \mathbb{Q}$ for all $v, w \in L$.

`IsIntegral(L): Lat -> BoolElt`

Return `true` if and only if L is an integral lattice, i.e., if and only if $(v, w) \in \mathbb{Z}$ for all $v, w \in L$.

`IsEven(L): Lat -> BoolElt`

Return `true` if and only if L is an even lattice, i.e., if and only if L is integral and $(v, v) \in 2\mathbb{Z}$ for all $v \in L$.

31.5.4 Base Ring and Base Change

`BaseRing(L): Lat -> Rng`

`CoefficientRing(L): Lat -> Rng`

Return the base ring of the lattice L, which is the ring over which the elements of L are represented. Note that lattices are always $\mathbb{Z}$-modules even if the base ring is not $\mathbb{Z}$; the base ring R is just defined to be the smallest ring over which the basis and inner product matrices can be represented. See the section on presentation of lattices at the beginning of the chapter for further discussion.

`CoordinateRing(L): Lat -> RngInt`

`Order(L): Lat -> RngInt`

Return the ring of coordinate coefficients for the lattice L. This currently will always return the integer ring $\mathbb{Z}$.

`ChangeRing(L, S): Lat, Rng -> Lat, Map`

`BaseChange(L, S): Lat, Rng -> Lat, Map`

`BaseExtend(L, S): Lat, Rng -> Lat, Map`

Given a lattice L, return the lattice L' obtained from coercing the entries of the basis and of the matrix for the inner product into the ring S, together with the homomorphism from L to L'. This will result in an error if any of the entries is not coercible into S. This function is only really useful for moving between real fields of varying precision (it is unnecessary and ineffectual to change a lattice from $\mathbb{Z}$ to $\mathbb{Q}$; see the section on presentation of lattices).

31.6 Construction of New Lattices

31.6.1 Sub- and Superlattices and Quotients

`sub<L | S>: Lat, List -> Lat`

Given a lattice L and a list S, construct the sublattice L' of L generated by the elements specified by the list S. Each term S_i of the list S must be an expression defining an object of one of the following types:

(a) an element of L or coercible into L,

(b) a set or sequence of elements coercible into L,

(c) a sublattice of L,

(d) a set or sequence of sublattices of L.

The constructor returns the sublattice L' and the inclusion homomorphism from L' into L.

`ext< L | S >: Lat, List -> Lat`

Given a lattice L lying inside $V = \mathbb{R}^n$ and a list S, construct the superlattice L' of L generated by L together with the elements specified by the list S. Each term S_i of the list S must be an expression defining an object of one of the following types:

(a) an element of V or coercible into V,

(b) a set or sequence of elements coercible into V,

(c) a sublattice of V,

(d) a set or sequence of sublattices of V.

The constructor returns the superlattice L' and the inclusion homomorphism from L into L'.

`T * L: AlgMatElt, Lat -> Lat`

`T * L: ModMatRng, Lat -> Lat`

Given a lattice L of rank m and an $l \times m$ integer matrix, construct the sublattice of L defined by the transformation matrix T, i.e., the lattice generated by the rows of the matrix obtained by multiplying the basis matrix of L from the left by T. The resulting lattice will have rank less than or equal to l.

`s * L: RngElt, Lat -> Lat`

`L * s: Lat, RngElt -> Lat`

Given a lattice L and a scalar s, construct the sublattice or superlattice of L obtained by multiplying the basis matrix of L by the scalar s.

`J * L: RngInt, Lat -> Lat`

`J * L: RngIntFracIdl, Lat -> Lat`

Given a lattice L and a fractional ideal J, return the lattice obtained by multiplying by the given ideal.

`L / s: Lat, RngElt -> Lat`

Given a lattice L and a scalar s, construct the sublattice or superlattice of L obtained by multiplying the basis matrix of L by the scalar $1/s$.

`quo< L | S >: Lat, List -> GrpAb, Map`

Given a lattice L and a list S, construct the quotient L/L', where L' is the sublattice of L generated by the elements of the list S. The elements of S must be the same as for the `sub<>` constructor. The quotient $Q := L/L'$ is constructed as an abelian group. As a second value the function returns the natural epimorphism $L \to Q$.

`L / S: Lat, Lat -> GrpAb, Map`

Given two lattices L and S such that S is a sublattice of L, construct the quotient $Q := L/S$ as an abelian group. As a second value the function returns the natural epimorphism $L \to Q$.

`ElementaryDivisors(A, B): Lat, Lat -> [ RngIntFracIdl ]`

The elementary divisors of the lattice A with respect to the lattice B. These are fractional ideals $D_1 \subseteq \ldots \subseteq D_n$ such that there exist pseudobases for A and B with the same basis vectors and coefficient ideals $J_{B,i} = D_i J_{A,i}$.

`Discriminant(A, B): Lat, Lat -> RngIntFracIdl`

The discriminant of the lattice A with respect to the lattice B. This is the product of the elementary divisors of A with respect to B.

`Index(L, S): Lat, Lat -> RngInt`

Given a lattice L and a sublattice S of L, return the index of S in L. This is the cardinality of the quotient L/S. If the index is infinite, zero is returned.

Example: Sublattices, Superlattices, and Quotients (ex-db15d9)

We demonstrate simple uses of the `sub`-, `ext`-, and `quo`-constructors.

```
> L := LatticeWithBasis(4, [1,2,3,4, 0,1,1,1, 0,1,3,5]);
> L;
Lattice of rank 3 and degree 4
Basis:
(1 2 3 4)
(0 1 1 1)
(0 1 3 5)
> E := ext<L | [1,0,0,0]>;
> E;
Lattice of rank 3 and degree 4
Basis:
( 1  0  0  0)
( 0  1  0 -1)
( 0  1  1  1)
> Index(E, L);
2
> Q, f := quo<E | L>;
> Q;
Abelian Group isomorphic to Z/2
Defined on 1 generator
Relations:
    2*Q.1 = 0
> f(E.1);
Q.1
```

31.6.2 Standard Constructions of New Lattices

The functions in this section enable one to construct new lattices from old ones using standard operations. Note that the functions will preserve the basis matrices of their arguments if possible (e.g., `DirectSum`), but if this is not possible the basis of the resulting lattice will be LLL-reduced (e.g., `+`, `meet`).

`Dual(L): Lat -> Lat`

```
Rescale: BoolElt                    Default: true
```

Let L be a lattice in $\mathbb{R}^n$ and let $V := \mathbb{R} \otimes_{\mathbb{Z}} L$ be the real vector space generated by L. Then the dual lattice $L^{\#}$ of L is defined by $L^{\#} := \{v \in V | (v, l) \in \mathbb{Z}\ \forall l \in L\}$. For an integral lattice L one always has $L \subseteq L^{\#}$. This function returns a rescaled version L' of the dual $L^{\#}$ by default so that the basis of L' is LLL-reduced and L' is an integral lattice and its Gram matrix is primitive (its entries are coprime). By setting the parameter `Rescale` to `false`, the rescaling can be suppressed and the proper dual lattice is returned.

`PartialDual(L, n): Lat, RngIntElt -> Lat`

```
Rescale: BoolElt                    Default: true
```

Given an integral lattice L and a positive integer n that divides the exponent e of the `DualQuotient` group of L, this function computes the nth partial dual of L. This is defined by pulling back $(e/n)\cdot g$ for generators g of the `DualQuotient` and intersecting with the lattice itself.

`DualBasisLattice(L): Lat -> Lat`

Let L be a lattice in $\mathbb{R}^n$ and let $V := \mathbb{R} \otimes_{\mathbb{Z}} L$ be the real vector space generated by L, then $L^{\#} := \{v \in V | (v, l) \in \mathbb{Z}\ \forall l \in L\}$ is the dual lattice of L. Let B be the basis matrix of L, M the inner product matrix of L, and F the Gram matrix of L. This function returns the dual $L^{\#}$ as the lattice with basis matrix $F^{-1}B$ and inner product matrix M (so that its Gram matrix is F^{-1}).

`DualQuotient(L): Lat -> GrpAb, Lat, Map`

Given an integral lattice L, construct the dual quotient Q of L, which is defined to be the finite abelian group $L^{\#}/L$ of order `Determinant(L)`, where $L^{\#}$ is the unscaled dual lattice of L (the lattice returned by `Dual` with the `Rescale` parameter set to `false`). This function returns three values:

(a) The dual quotient Q.

(b) The unscaled dual lattice $L^{\#}$.

(c) The natural epimorphism $\phi : L^{\#} \to Q$ whose kernel is L.

`EvenSublattice(L): Lat -> Lat, Map`

Given an integral lattice L, construct its maximal even sublattice together with the natural embedding into L.

Example: Dual Quotient of a Lattice (ex-4c9c8e)

```
> L := Lattice(LatticeDatabase(),"LAMBDA29");
> Dimension(L);
29
> IsEven(L);
true
> IsEven(Dual(L));
false;
> G := DualQuotient(L);
> Exponent(G);
8
> Factorization(Determinant(L));
[ <2, 31> ]
> PartialDual(L,1) eq L;
true
> Factorization(Determinant(PartialDual(L, 2)));
[ <2, 54> ]
> Factorization(Determinant(PartialDual(L, 4)));
[ <2, 73> ]
> Factorization(Determinant(PartialDual(L, 8)));
[ <2, 56> ]
> Factorization(Determinant(Dual(L)));
[ <2, 56> ]
```

`L + M: Lat, Lat -> Lat`

Given compatible lattices L and M, construct the lattice generated by their union.

`L meet M: Lat, Lat -> Lat`

Given compatible lattices L and M, construct their intersection $L \cap M$.

`DirectSum(L, M): Lat, Lat -> Lat`

`OrthogonalSum(L, M): Lat, Lat -> Lat`

Given lattices L and M, construct their orthogonal sum which is their direct sum with inner product being the orthogonal sum of the inner products of L and M.

`OrthogonalDecomposition(L): Lat -> [Lat]`

`OrthogonalDecomposition(L, F): Lat, [Mtrx] -> [Lat]`

Given a lattice L, construct the sequence of indecomposable orthogonal summands composing L. Additional bilinear forms can be given in F. In this case the decomposition will be orthogonal wrt. these forms as well.

`OrthogonalDecomposition(F): [Mtrx] -> [* Mtrx *], [* [Mtrx] *]`

`Optimize: BoolElt` `Default: false`

Given a sequence of bilinear forms F, where the first form is positive definite, returns the basis matrices $B_1, \ldots, B_s$ of the indecomposable orthogonal summands of the standard lattice $\mathbb{Z}^n$ wrt. the forms in F. The second return value is a list of s sequences. The i-th sequence contains the forms of F wrt. to basis described by B_i. If `Optimize` is set, then the basis matrices will be LLL reduced wrt. the first form in F.

`TensorProduct(L, M): Lat, Lat -> Lat`

Given two lattices L and M, construct their tensor product with inner product given by the Kronecker product of the matrices defining the inner products of L and M.

`ExteriorSquare(L): Lat -> Lat`

Given a lattice L, construct its exterior square, generated by the skew tensors in $L \otimes L$. The inner product is inherited from the inner product of the tensor square of the vector space containing L and the exterior square lattice lies inside the tensor square of the lattice.

`SymmetricSquare(L): Lat -> Lat`

Given a lattice L, construct its symmetric square, generated by the symmetric tensors in $L \otimes L$. The inner product is inherited from the inner product of the tensor square of the vector space containing L and the symmetric square lattice lies inside the tensor square of the lattice.

`PureLattice(L): Lat -> Lat`

Given a lattice L of degree n with integral or rational entries, return the pure lattice $P = (\mathbb{Q} \otimes L) \cap \mathbb{Z}^n$ of L. The pure lattice P generates the same subspace in $\mathbb{Q}^n$ over $\mathbb{Q}$ that L does but the elementary divisors of its basis matrix are trivial.

`IntegralBasisLattice(L): Lat -> Lat, RngIntElt`

Given an exact lattice L, return the lattice obtained from L by multiplying the basis by the smallest positive scalar S so that the resulting basis is integral, and S.

31.7 Reduction of Matrices and Lattices

The functions in this section perform reduction of lattice bases. For each reduction algorithm there are three functions: a function which takes a basis matrix, a function which takes a Gram matrix and a function which takes a lattice.

31.7.1 LLL Reduction

The Lenstra-Lenstra-Lovász algorithm [LLL82] was first described in 1982 and was immediately used by the authors to provide a polynomial-time algorithm for factoring integer polynomials, for solving simultaneous diophantine equations, and for solving the integer programming problem. It very quickly received much attention and in the last 25 years has found an incredible range of applications, in such areas as computer algebra, cryptography, optimisation, algorithmic number theory, group theory, etc. However, the original LLL algorithm is mainly of theoretical interest, since, although it has polynomial time complexity, it remains quite slow in practice. Rather, floating-point variants are used, where

the underlying Gram-Schmidt orthogonalisation is performed with floating-point arithmetic instead of rational arithmetic (which produces huge numerators and denominators). Most of these floating-point variants are heuristic, but here we use the provable Nguyen-Stehlé algorithm [NS09a] (see also [Ste09] for more details about the implementation). Let $\delta \in (1/4, 1]$ and $\eta \in [1/2, \sqrt{\delta})$. An ordered set of d linearly independent vectors $b_1, b_2, \ldots, b_d \in \mathbb{R}^n$ is said to be (δ, η)-LLL-reduced if the two following conditions are satisfied:

(a) For any $i > j$, we have $|\mu_{i,j}| \leq \eta$,

(b) For any $i < d$, we have $\delta \|b_i^*\|^2 \leq \|b_{i+1}^* + \mu_{i+1,i} b_{i+1}^*\|^2$,

where $\mu_{i,j} = (b_i, b_j^*)/\|b_j^*\|^2$ and b_i^* is the i-th vector of the Gram-Schmidt orthogonalisation of $(b_1, b_2, \ldots, b_d)$. LLL-reduction classically refers to the case where $\eta = 1/2$ and $\delta = 3/4$ since it was the choice of parameters originally made in [LLL82], but the closer that η and δ are to $1/2$ and 1, respectively, the more reduced the lattice basis should be. In the classical LLL algorithm, the polynomial-time complexity is guaranteed as long as $\delta < 1$, whereas the floating-point LLL additionally requires that $\eta > 1/2$. A (δ, η)-LLL-reduced basis $(b_1, b_2, \ldots, b_d)$ of a lattice L has the following useful properties:

(a) $\|b_1\| \leq \left(\delta - \eta^2\right)^{-(d-1)/4} (\det L)^{1/d}$,

(b) $\|b_1\| \leq (\delta - \eta^2)^{-(d-1)/2} \min_{b \in L \setminus \{0\}} \|b\|$,

(c) $\prod_{i=1}^{d} \|b_i\| \leq (\delta - \eta^2)^{-d(d-1)/4} (\det L)$,

(d) For any $j < i$, $\|b_j^*\| \leq (\delta - \eta^2)^{(j-i)/2} \|b_i^*\|$.

The default Magma parameters are $\delta = 0.75$ and $\eta = 0.501$, so that $(\delta - \eta^2)^{-1/4} < 1.190$ and $(\delta - \eta^2)^{-1/2} < 1.416$. The four previous bounds can be reached, but in practice one usually obtains better bases. It is possible to obtain lattice bases satisfying these four conditions without being LLL-reduced, by using the so-called Siegel condition [Akh02]; this useful variant, though available, is not the default one. Internally, the LLL routine may use up to eight different LLL variants, one of them being de Weger's exact integer method [dW87], and the seven others relying upon diverse kinds of floating-point arithmetic. All but one of these variants are heuristic and implement diverse ideas from [SE94], Victor Shoup's NTL [Sho], and [NS]. The heuristic variants possibly loop forever, but when that happens it is hopefully detected and a more reliable variant is used. For a given input, the LLL routine tries to find out which variant should be the fastest, and may eventually call other variants before the end of the execution: either because it deems that the current variant loops forever or that a faster variant could be used with the thus-far reduced basis. This machinery remains unknown to the user (except when the LLL verbose mode is activated) and makes the Magma LLL routine the fastest currently available, with the additional advantage of being fairly reliable. The floating-point variants essentially differ on the two following points:

(a) whether or not the matrix basis computations are performed with the help of a complete knowledge of the Gram matrix of the vectors (the matrix of the pairwise inner products),

(b) the underlying floating-point arithmetic.

In theory, to get a provable variant, one needs the Gram matrix and arbitrary precision floating-point numbers (as provided by the MPFR-based [Pro] Magma real numbers). In practice, to get an efficient variant, it is not desirable not to maintain the exact knowledge of the Gram matrix (maintaining the Gram matrix represents asymptotically a large constant fraction of the total computational cost), while it is desirable to use the machine processor floating-point arithmetic (for instance, C doubles). Three of the seven variants use the Gram matrix, while the four others do not. In each group (either the three or the remaining four), one variant relies on arbitrary precision floating-point arithmetic, another on C doubles, and another on what we call doubles with extended exponent. These last variants are important because they are almost as fast as the variants based on C doubles, and can be used for much wider families of inputs: when the initial basis is made of integers larger than approximately 500 bits, overflows can occur with the C doubles. To be precise, a "double with extended exponent" is a C double with a C integer, the last one being the extended exponent. So far, we have described six variants. The seventh does not rely on the Gram matrix and is based on an idea similar to "doubles with extended exponents". The difference is that for a given vector of dimension n, instead of having n pairs of doubles and integers, one has n doubles and only one integer: the extended exponent is "factorised". This variant is quite often the one to use in practice, because it is reasonably efficient and not too unreliable.

Important warning. By default, the `LLL` routine is entirely reliable. This implies the default provable variant can be *much* slower than the heuristic one. By setting `Proof` to `false`, a significant run-time improvement can be gained without taking much risk: even in that case, the `LLL` routine remains more reliable than any other implementation based on floating-point arithmetic. For any application where guaranteed LLL-reduction is not needed, we recommend setting `Proof` to `false`.

Recommended usage. The formal description of the function `LLL` below explains all the parameters in detail, but we first note here some common situations which arise. The desired variant of LLL is often not the default one. Since the LLL algorithm is used in many different contexts and with different output requirements, it seems impossible to define a natural default variant, and so using the LLL routine efficiently often requires some tuning. Here we consider three main-stream situations.

- It may be desired to obtain the main LLL inequalities (see the introduction of this subsection), without paying much attention to the δ and η reduction parameters. In this case one should activate the `Fast` parameter, possibly with the verbose mode to know afterwards which parameters have been used.
- It may be desired to have the main LLL inequalities for a given pair of parameters (δ, η). In this case one should set the parameters `Delta` and `Eta` to the desired values, and set `SwappingCondition` to `"Siegel"`.
- It may be desired to compute a very well LLL-reduced basis. In this case one should set `Delta` to 0.9999, `Eta` to 0.5001, and possibly also activate the `DeepInsertion` option.

In any case, if you want to be absolutely sure of the quality of the output, you need to keep the `Proof` option activated.

Example: Recommended Uses of LLL `(ex-5c6218)`

This example is meant to show the differences between the three main recommended usages above. Assume that we are interested in the lattice generated by the rows of the matrix B defined as follows.

```
> R:=RealField(5);
> B:=RMatrixSpace(IntegerRing(), 50, 51) ! 0;
> for i := 1 to 50 do B[i][1] := RandomBits(10000); end for;
> for i := 1 to 50 do B[i][i+1] := 1; end for;
```

The matrix B is made of 100 vectors of length 101. Each entry of the first column is approximately 10000 bits long. Suppose first that we use the default variant.

```
> time C:=LLL(B:Proof:=false);
Time: 11.300
> R!(Norm (C[1]));
5.1959E121
```

The output basis is $(0.75, 0.501)$-reduced. Suppose now that we only wanted to have a basis that satisfies the main LLL properties with $\delta = 0.75$ and $\eta = 0.501$. Then we could have used the Siegel swapping condition.

```
> time C:=LLL(B:Proof:=false, SwapCondition:="Siegel");
Time: 10.740
> R!(Norm (C[1]));
6.6311E122
```

Notice that the quality of the output is quite often worse with the Siegel condition than with the Lovász condition, but the main LLL properties are satisfied anyway. Suppose now that we want a very well reduced basis. Then we fix δ and η close to 1 and 0.5 respectively.

```
> time C:=LLL(B:Proof:=false, Delta:=0.9999, Eta:=0.5001);
Time: 19.220
```

(continues on next page)

(continued from previous page)

```
> R!(Norm (C[1]));
1.8056E121
```

This is of course more expensive, but the first output vector is significantly shorter. Finally, suppose that we only wanted to "shorten" the entries of the input basis, very efficiently and without any particular preference for the reduction factors δ and η. Then we could have used the `Fast` option, that tries to choose such parameters in order to optimize the running-time.

```
> time C:=LLL(B:Proof:=false, Fast:=1);
Time: 8.500
> R!(Norm (C[1]));
6.2746E121
```

By activating the verbose mode, one can know for which parameters the output basis is reduced.

```
> SetVerbose ("LLL", 1);
> C:=LLL(B:Proof:=false, Fast:=1);
[...]
The output basis is (0.830,0.670)-reduced
```

`LLL(X): ModMatRngElt -> ModMatRngElt, AlgMatElt, RngIntElt`

`LLL(X): AlgMatElt -> AlgMatElt, AlgMatElt, RngIntElt`

```
Al              : MonStgElt                 Default: "New"
Proof           : BoolElt                   Default: true
Method          : MonStgElt                 Default: "FP"
Delta           : RngElt                    Default: 0.75
Eta             : RngElt                    Default: 0.501
InitialSort     : BoolElt                   Default: false
FinalSort       : BoolElt                   Default: false
StepLimit       : RngIntElt                 Default: 0
TimeLimit       : RngElt                    Default: 0.0
NormLimit       : RngIntElt                 Default:
UseGram         : BoolElt                   Default: false
DeepInsertions: BoolElt                     Default: false
EarlyReduction: BoolElt                     Default: false
SwapCondition : MonStgElt                   Default: "Lovasz"
Fast            : RngIntElt                 Default: 0
Weight          : SeqEnum                   Default: [0,...,0]
```

Given a matrix X belonging to the matrix module $S = \mathrm{Hom}_R(M, N)$ or the matrix algebra $S = M_n(R)$, where R is a subring of the real field, compute a matrix Y whose non-zero rows are a LLL-reduced basis for the $\mathbb{Z}$-lattice spanned by the rows of X (which need not be $\mathbb{Z}$-linearly independent). The LLL function returns three values:

(a) A LLL-reduced matrix Y in S whose rows span the same lattice ($\mathbb{Z}$-module) as that spanned by the rows of X.

(b) A unimodular matrix T in the matrix ring over $\mathbb{Z}$ whose degree is the number of rows of X such that $TX = Y$;

(c) The rank of X.

Note that the returned transformation matrix T is *always* over the ring of integers $\mathbb{Z}$, even if R is not $\mathbb{Z}$. The input matrix X does not need to have linearly independent rows: this function performs a floating-point variant of what is usually known as the MLLL algorithm of M. Pohst [Poh87]. By default the rows of the returned matrix Y are sorted so that all the zero rows are at the bottom. The non-zero vectors are sorted so that they form a LLL-reduced basis (except when `FinalSort` is

`true`; see below). A detailed description of the parameters now follows. See the discussion and the example above this function for recommended parameters for common cases. By default, the `Proof` option is set to `true`. It means that the result is guaranteed. It is possible, and usually faster, to switch off this option: it will perform the same calculations, without checking that the output is indeed reduced by using the L^2 algorithm. For the vast majority of cases, we recommend setting `Proof` to `false`. By default the δ and η constants used in the LLL conditions are set respectively to 0.75 and 0.501 (except when `Method` is `"Integral"`, see below). The closer δ and η are to 1 and 0.5, respectively, the shorter the output basis will be. In the original description of the LLL algorithm, δ is 0.75 and η is 0.5. Making δ and η vary can have a significant influence on the running time. If `Method` is `"Integral"`, then by default η is set to 0.5 and δ to 0.99. With this value of `Method`, the parameter δ can be chosen arbitrarily in $(0.25, 1]$, but when δ is 1, the running time may increase dramatically. If `Method` is not `"Integral"`, then δ may be chosen arbitrarily in $(0.25, 1)$, and η may be chosen arbitrarily in $(0.5, \sqrt{\delta})$. Notice that the parameters δ and η used internally will be slightly larger and smaller than the given ones, respectively, to take floating-point inaccuracies into account and to ensure that the output basis will indeed be reduced for the expected parameters. In all cases, the output basis will be (δ, η)-LLL reduced. For matrices over $\mathbb{Z}$ or $\mathbb{Q}$, there are two main methods for handling the Gram-Schmidt orthogonalisation variables used in the algorithm: the Nguyen-Stehlé floating-point method (called L^2) and the exact integral method described by de Weger in [dW87]. The Nguyen-Stehlé algorithm is implemented as described in the article, along with a few faster heuristic variants. When `Method` is `"FP"`, these faster variants will be tried first, and when they are considered as misbehaving, they will be followed by the proved variant automatically. When `Method` is `"L2"`, the provable L^2 algorithm is used directly. Finally, when `Method` is `"Integral"`, the De Weger integral method is used. In this case, the `DeepInsertions`, `EarlyReduction`, `Siegel SwapCondition`, `NormLimit`, `Fast`, and `Weight` options are unavailable (the code is older and has not been updated for these). Furthermore, the default value for `Eta` is then 0.5. The default `FP` method is nearly always *very* much faster than the other two methods. By default, it is possible (though it happens very infrequently) that the returned basis is not of the expected quality. In order to be sure that the output is indeed correct, `Method` can be set to either `"Integral"` or `"L2"`. Since V2.26, a parallel multi-threaded version of the integral method (selected by `Method := "Integral"`) is now supported. One can call `SetNthreads` beforehand to specify the number of threads to be used. For inputs in larger dimension, this may lead to a significant speedup.

The parameter `InitialSort` specifies whether the vectors should be sorted by length before the LLL reduction. When `FinalSort` is `true`, the vectors are sorted by increasing length (and alphabetical order in case of equality) after the LLL reduction (this parameter was called `Sort` before V2.13). The resulting vectors may therefore not be strictly LLL-reduced, because of the permutation of the rows. The parameter `UseGram` specifies that for the floating-point methods (`L2` and `FP`) the computation should be performed by computing the Gram matrix F, using the `LLLGram` algorithm below, and by updating the basis matrix correspondingly. Magma will automatically do this internally for the floating-point method if it deems that it is more efficient to do so, so this parameter just allows one to stipulate that this method should or should not be used. For the integral method, using the Gram matrix never improves the computation, so the value of this parameter is simply ignored in this case. Setting the parameter `StepLimit` to a positive integer s will cause the function to terminate after s steps of the main loop. Setting the parameter `TimeLimit` to a positive real number t will cause the function to terminate after the algorithm has been executed for t seconds (process user) time. Similarly, setting the parameter `NormLimit` to a non-negative integer N will cause the algorithm to terminate after a vector of norm less or equal to N has been found. In such cases, the current reduced basis is returned, together with the appropriate transformation matrix and an upper bound for the rank of the matrix. Nothing precise can then be said exactly about the reduced basis (it will not be LLL-reduced in general of course) but will at least be reduced to a certain extent. When the value of the parameter `DeepInsertions` is `true`, Schnorr-Euchner's deep insertion algorithm is used [SE94]. It usually provides better bases, but can take significantly more time. In practice, one may first LLL-reduce the basis and then use the deep insertion algorithm on the computed basis. When the parameter `EarlyReduction` is `true` and when any of the two floating-point methods are used, some early reduction steps are inserted inside the execution of the LLL algorithm. This sometimes makes the entries of the basis smaller very quickly. It occurs in particular for lattice bases built from minimal polynomial or integer relation detection problems. The speed-up is sometimes dramatic, especially if the reduc-

tion in length makes it possible to use C integers for the basis matrix (instead of multiprecision integers) or C doubles for the floating-point calculations (instead of multiprecision floating-point numbers or doubles with additional exponent). The parameter `SwapCondition` can be set either to `"Lovasz"` (default) or to `"Siegel"`. When its value is `"Lovasz"`, the classical Lovász swapping condition is used, whereas otherwise the so-called *Siegel* condition is used. The Lovász condition tests whether the inequality $\|b_i^* + \mu_{i,i-1}b_{i-1}^*\|^2 \geq \delta\|b_{i-1}^*\|^2$ is satisfied or not, whereas in the Siegel case the inequality $\|b_i^*\|^2 \geq (\delta - \eta^2)\|b_{i-1}^*\|^2$ is used (see [Akh02] for more details). In the Siegel case, the execution may be significantly faster. Though the output basis may not be LLL-reduced, the classical LLL inequalities (see the introduction of this subsection) will be fulfilled. When the option `Fast` is set to 1 or 2, all the other parameters may be changed internally to end the execution as fast as possible. Even if the other parameters are not set to the default values, they may be ignored. This includes the parameters `Delta`, `Eta`, and `SwapCondition`, so that the output basis may not be LLL-reduced. However, if the verbose mode is activated, the chosen values of these parameters will be printed, so that the classical LLL inequalities (see the introduction of this subsection) may be used afterwards. When `Fast` is set to 2, the chosen parameters are such that the classical LLL inequalities will be at least as strong as for the default parameters `Delta` and `Eta`. If `Weight` is the list of integers $[x_1, \ldots, x_n]$ (where n is the degree of the lattice), then the LLL algorithm uses a weighted Euclidean inner product: the inner product of the vectors $(v_1, v_2, \ldots, v_n)$ and $(w_1, w_2, \ldots, w_n)$ is defined to be $\sum_{i=1}^{n} \left(v_i \cdot w_i \cdot 2^{2x_i}\right)$. By default, the `Al` parameter is set to `New`. If it is set to `Old`, then the former Magma LLL is used (prior to V2.13); notice that in this case, the execution is not guaranteed to finish and that the output basis may not be reduced for the given LLL parameters. This variant is not maintained anymore.

Note 1: If the input basis has many zero entries, then try to place them in the last columns; this will slightly improve the running time.

Note 2: If the elementary divisors of the input matrix are fairly small, relative to the size of the matrix and the size of its entries, then it may be very much quicker to reduce the matrix to Hermite form first (using the function `HermiteForm`) and then to LLL-reduce this matrix. This case can arise: for example, when one has a "very messy" matrix for which it is known that the lattice described by it has a relatively short basis.

Note 3: Sometimes, the `EarlyReduction` variant can significantly decrease the running time. Since V2.21 (2014), Magma also supports an analogue for LLL-reduction for matrices over $K[x]$. The algorithm is similar to that described in [Pau98]. Note that unlike the integer case, the LLL-reduction of a matrix in the $K[x]$ case is always unique, with the rows sorted by degree.

`BasisReduction(X): ModMatRngElt -> ModMatRngElt, AlgMatElt, RngIntElt`

`BasisReduction(X): AlgMatElt -> AlgMatElt, AlgMatElt, RngIntElt`

This is a shortcut for `LLL(X:Proof:=false)`.

`LLLGram(F): ModMatRngElt -> ModMatRngElt, AlgMatElt, RngIntElt`

`LLLGram(F): AlgMatElt -> AlgMatElt, AlgMatElt, RngIntElt`

`Isotropic: BoolElt` `Default: false`

Given a symmetric matrix F belonging to the matrix module $S = \mathrm{Hom}_R(M, M)$ or the matrix algebra $S = M_n(R)$, where R is a subring of the real field, so that $F = XX^{tr}$ for some matrix X over the real field, compute a matrix G which is the Gram matrix corresponding to a LLL-reduced form of the matrix X. The rows of the corresponding generator matrix X need not be $\mathbb{Z}$-linearly independent in which case F will be singular. This function returns three values:

(a) A LLL-reduced Gram matrix G in S of the Gram matrix F;

(b) A unimodular matrix T in the matrix ring over $\mathbb{Z}$ whose degree is the number of rows of F such that $G = TFT^{tr}$.

(c) The rank of F (which equals the dimension of the lattice generated by X in the definite case).

The options available are the same as for the `LLL` routine, except of course the `UseGram` option that has no sense here. The `Weight` parameter should not be used either. The routine can be used with

any symmetric matrix (possibly not definite nor positive): Simon's indefinite LLL variant (which puts absolute values on the Lovász condition) is used (see [Sim05b]). When the option `Isotropic` is true, some attempt to deal with isotropic vectors is made. (This is only relevant when working on an indefinite Gram matrix). In particular, if the determinant is squarefree, hyperbolic planes are split off iteratively.

`LLLBasisMatrix(L): Lat -> ModMatElt, AlgMatElt`

Given a lattice L with basis matrix B, return *the LLL basis matrix* B' of L, together with the transformation matrix T such that $B' = TB$. The LLL basis matrix B' is simply defined to be a LLL-reduced form of B; it is stored in L when computed and subsequently used internally by many lattice functions. The LLL basis matrix will be created automatically internally as needed with $\delta = 0.999$ by default (note that this is different from the usual default of 0.75); by the use of parameters to this function one can ensure that the LLL basis matrix is created in a way which is different to the default. The parameters (not listed here again) are the same as for the function `LLL` (q.v.), except that the limit parameters are illegal.

`LLLGramMatrix(L): Lat -> AlgMatElt, AlgMatElt`

Given a lattice L with Gram matrix F, return the *LLL Gram matrix* F' of L, together with the transformation matrix T such that $F' = TFT^{tr}$. F' is simply defined to be $B'(B')^{tr}$, where B' is the LLL basis matrix of L—see the function `LLLBasisMatrix`. The parameters (not listed here again) are the same as for the function `LLL` (q.v.), except that the limit parameters are illegal.

`LLL(L): Lat -> Lat, AlgMatElt`

Given a lattice L with basis matrix B, return a new lattice L' with basis matrix B' and a transformation matrix T so that L' is equal to L but B' is LLL-reduced and $B' = TB$. Note that the inner product used in the LLL computation is that given by the inner product matrix of L (so, for example, the resulting basis may not be LLL-reduced with respect to the standard Euclidean norm). The LLL basis matrix of L is used, so calling this function with argument L is completely equivalent (ignoring the second return value) to the invocation `LatticeWithBasis(LLLBasisMatrix(L), InnerProductMatrix(L))`. The parameters (not listed here again) are the same as for the function `LLL` (q.v.), except that the limit parameters are illegal. The *`BasisReduction`* shortcut to turn off the `Proof` option is also available.

`BasisReduction(L): Lat -> Lat, AlgMatElt`

This is a shortcut for `LLL(L:Proof:=false)`.

`SetVerbose("LLL", v): MonStgElt, RngIntElt`

(Procedure.) Set the verbose printing level for the LLL algorithm to be v. Currently the legal values for v are `true`, `false`, 0, 1, 2, and 3 (`false` is the same as 0, and `true` is the same as 1). The three non-zero levels notify when the maximum LLL-reduced rank of the LLL algorithm increases, level 2 also prints the norms of the reduced vectors at such points, and level 3 additionally gives some current status information every 15 seconds.

Example: Extended GCD via LLL (ex-471339)

We demonstrate how the LLL algorithm can be used to find very good multipliers for the extended GCD of a sequence of integers. Given a sequence $Q = [x_1, \ldots, x_n]$ of integers we wish to find the GCD g of Q and integers m_i for $1 \leq i \leq m$ such that $g = m_1 \cdot x_1 + \ldots + m_n \cdot x_m$. For this example we set Q to a sequence of $n = 10$ integers of varying bit lengths (to make the problem a bit harder).

```
> Q := [ 67015143, 248934363018, 109210, 25590011055, 74631449,
>        10230248, 709487, 68965012139, 972065, 864972271 ];
> n := #Q;
> n;
10
```

We next choose a scale factor S large enough and then create the $n \times (n+1)$ matrix $X = [I_n|C]$ where C is the column vector whose i-th entry is $S \cdot Q[i]$.

```
> S := 100;
> X := RMatrixSpace(IntegerRing(), n, n + 1) ! 0;
> for i := 1 to n do X[i][i + 1] := 1; end for;
> for i := 1 to n do X[i][1] := S * Q[i]; end for;
> X;
[6701514300     1 0 0 0 0 0 0 0 0 0]
[24893436301800 0 1 0 0 0 0 0 0 0 0]
[10921000       0 0 1 0 0 0 0 0 0 0]
[2559001105500  0 0 0 1 0 0 0 0 0 0]
[7463144900     0 0 0 0 1 0 0 0 0 0]
[1023024800     0 0 0 0 0 1 0 0 0 0]
[70948700       0 0 0 0 0 0 1 0 0 0]
[6896501213900  0 0 0 0 0 0 0 1 0 0]
[97206500       0 0 0 0 0 0 0 0 1 0]
[86497227100    0 0 0 0 0 0 0 0 0 1]
```

Finally, we compute a LLL-reduced form L of X.

```
> L := LLL(X);
> L;
[   0    0    1    0  -15   -6    3    1    2   -3   -3]
[   0   -3    5   -3  -11   -1    0    8  -14    4    3]
[   0   -5   -2    2   -2   14    8   -6    8    1   -4]
[   0    6    1   -3  -10   -3  -14    5    0    8    8]
[   0   -1   -2    0   -2  -13    8    6    8   10   -2]
[   0   -9   -3  -11    5    7   -1    1    9  -11   -2]
[   0   16    0    3    3    9   -3    0   -1   -4  -11]
[   0   -6    1  -16    4   -1    6   -6   -5    7   -7]
[   0    9   -3  -10    7   -3    1   -7    8    0   18]
[-100   -3   -1   13   -1   -4    2    3    4    5   -1]
```

Notice that the large weighting on the first column forces the LLL algorithm to produce as many vectors as possible at the top of the matrix with a zero entry in the first column (since such vectors will have shorter norms than any vector with a non-zero entry in the first column). Thus the GCD of the entries in Q is the entry in the bottom left corner of L divided by S, so this must be 1 or -1. The last n entries of the last row of L gives a sequence of multipliers M for the extended GCD algorithm. Also, taking the last n entries of each of the first $n-1$ rows of L gives independent null-relations for the entries of Q (i.e., the kernel of the corresponding column matrix). We check that M gives a correct sequence of multipliers.

```
> M := Eltseq(L[10])[2 .. n+1]; M;
[ 3, 1, -13, 1, 4, -2, -3, -4, -5, 1 ]
> &+[Q[i]*M[i]: i in [1 .. n]];
-1
```

31.7.2 Pair Reduction

`PairReduce(X): ModMatRngElt -> ModMatRngElt, AlgMatElt`

`PairReduce(X): AlgMatElt -> AlgMatElt, AlgMatElt`

Given a matrix X belonging to the matrix module $S = \mathrm{Hom}_R(M, N)$ or the matrix algebra $S = M_n(R)$, where R is $\mathbb{Z}$ or $\mathbb{Q}$, compute a matrix Y whose rows form a pairwise reduced basis for the $\mathbb{Z}$-lattice spanned by the rows of X. Being pairwise reduced (i.e., $2|(v, w)| \leq \min(\|v\|, \|w\|)$ for all pairs of basis vectors) is a much simpler criterion than being LLL-reduced, but often yields sufficiently good results very quickly. It can also be used as a preprocessing for LLL or in alternation with LLL to obtain better bases. The rows of X need not be Z-linearly independent. This function returns two values:

(a) The pairwise reduced matrix Y row-equivalent to X as a matrix in S;

(b) A unimodular matrix T in the matrix ring over $\mathbb{Z}$ whose degree is the number of rows of X such that $TX = Y$.

`PairReduceGram(F): ModMatRngElt -> ModMatRngElt, AlgMatElt, RngIntElt`

`PairReduceGram(F): AlgMatElt -> AlgMatElt, AlgMatElt, RngIntElt`

`Check: BoolElt` `Default: false`

Given a symmetric positive semidefinite matrix F belonging to the matrix module $S = \mathrm{Hom}_R(M, M)$ or the matrix algebra $S = M_n(R)$, where R is $\mathbb{Z}$ or $\mathbb{Q}$, so that $F = XX^{tr}$ for some matrix X over the real field, compute a matrix G which is the Gram matrix corresponding to a pairwise reduced form of the matrix X. The rows of the corresponding matrix X need not be Z-linearly independent. This function returns two values:

(a) The pairwise reduced Gram matrix G of F as a matrix in S;

(b) A unimodular matrix T in the matrix ring over $\mathbb{Z}$ whose degree is the number of rows of F which gives the corresponding transformation: $G = TFT^{tr}$. The matrix F must be a symmetric positive semidefinite matrix; if it is not the results are unpredictable. By default, Magma does not check this since it may be expensive in higher dimensions and in many applications will be known a priori. The checking can be invoked by setting `Check := true`.

`PairReduce(L): Lat -> Lat, AlgMatElt`

Given a lattice L with basis matrix B, return a new lattice L' with basis matrix B' and a transformation matrix T so that L' is equal to L but B' is pairwise reduced and $B' = TB$. Note that the inner product used in the pairwise reduction computation is that given by the inner product matrix of L (so, for example, the resulting basis may not be pairwise reduced with respect to the standard Euclidean norm).

31.7.3 Seysen Reduction

`Seysen(X): ModMatRngElt -> ModMatRngElt, AlgMatElt`

`Seysen(X): AlgMatElt -> AlgMatElt, AlgMatElt`

Given a matrix X belonging to the matrix module $S = \mathrm{Hom}_R(M, N)$ or the matrix algebra $S = M_n(R)$, where R is $\mathbb{Z}$ or $\mathbb{Q}$, compute a matrix Y whose rows form a Seysen-reduced basis for the $\mathbb{Z}$-lattice spanned by the rows of X. The rows of X need not be Z-linearly independent. The Seysen-reduced matrix Y is such that the entries of the corresponding Gram matrix $G = YY^{tr}$ and its inverse G^{-1} are simultaneously reduced. This function returns two values:

(a) The Seysen-reduced matrix Y corresponding to X as a matrix in S;

(b) A unimodular matrix T in the matrix ring over $\mathbb{Z}$ whose degree is the number of rows of X such that $TX = Y$.

`SeysenGram(F): ModMatRngElt -> ModMatRngElt, AlgMatElt, RngIntElt`

`SeysenGram(F): AlgMatElt -> AlgMatElt, AlgMatElt, RngIntElt`

`Check: BoolElt` `Default: false`

Given a symmetric positive semidefinite matrix F belonging to the matrix module $S = \mathrm{Hom}_R(M, M)$ or the matrix algebra $S = M_n(R)$, where R is $\mathbb{Z}$ or $\mathbb{Q}$, so that $F = XX^{tr}$ for some matrix X over the real field, compute a matrix G which is the Gram matrix corresponding to a Seysen-reduced form of the matrix X. The rows of the corresponding matrix X need not be Z-linearly independent. The Seysen-reduced Gram matrix G is such that the entries of G and its inverse G^{-1} are simultaneously reduced. This function returns two values:

(a) The Seysen-reduced Gram matrix G of F as a matrix in S;

(b) A unimodular matrix T in the matrix ring over $\mathbb{Z}$ whose degree is the number of rows of F which gives the corresponding transformation: $G = TFT^{tr}$. The matrix F must be a symmetric positive semidefinite matrix; if it is not the results are unpredictable. By default, Magma does not check this since it may be expensive in higher dimensions and in many applications will be known a priori. The checking can be invoked by setting `Check := true`.

`Seysen(L): Lat -> Lat, AlgMatElt`

Given a lattice L with basis matrix B, return a new lattice L' with basis matrix B' and a transformation matrix T so that L' is equal to L but B' is Seysen-reduced and $B' = TB$. Note that the inner product used in the Seysen-reduction computation is that given by the inner product matrix of L (so, for example, the resulting basis may not be Seysen-reduced with respect to the standard Euclidean norm). The effect of the reduction is that the basis of L' and the dual basis of L' will be simultaneously reduced.

Example: Seysen Reduction of the Leech Lattice `(ex-1751af)`

We demonstrate how the function `Seysen` can be used on the Gram matrix of the Leech lattice to obtain a gram matrix S for the lattice so that both S and S^{-1} are simultaneously reduced. Note that all three reduction methods yield a basis of vectors of minimal length, but the Seysen reduction also has a dual basis of vectors of norm 4. This is of interest in representation theory, for example, as the entries of the inverse of the Gram matrix control the size of the entries in a representation on the lattice.

```
> F := GramMatrix(Lattice("Lambda", 24));
> [ F[i][i] : i in [1..24] ];
[ 8, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4 ]
> [ (F^-1)[i][i] : i in [1..24] ];
[ 72, 8, 12, 8, 10, 4, 4, 8, 8, 4, 4, 8, 4, 4, 4, 4, 4, 4, 4, 4, 8, 4, 4, 8 ]
> L := LLLGram(F);
> P := PairReduceGram(F);
> S := SeysenGram(F);
> [ L[i][i] : i in [1..24] ];
[ 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4 ]
> [ P[i][i] : i in [1..24] ];
[ 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4 ]
> [ S[i][i] : i in [1..24] ];
[ 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4 ]
> [ (L^-1)[i][i] : i in [1..24] ];
[ 6, 4, 8, 4, 32, 4, 4, 8, 18, 4, 4, 8, 8, 8, 12, 4, 6, 4, 20, 4, 8, 4, 14, 4 ]
> [ (P^-1)[i][i] : i in [1..24] ];
[ 72, 40, 12, 8, 10, 4, 4, 8, 8, 4, 4, 8, 4, 4, 4, 4, 4, 4, 4, 4, 8, 4, 4, 8 ]
> [ (S^-1)[i][i] : i in [1..24] ];
[ 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4 ]
```

31.7.4 HKZ Reduction

An ordered set of d linearly independent vectors $b_1, b_2, \ldots, b_d \in \mathbb{R}^n$ is said to be Hermite-Korkine-Zolotarev-reduced (HKZ-reduced for short) if the three following conditions are satisfied:

(a) For any $i > j$, we have $|\mu_{i,j}| \leq 0.501$,

(b) The vector b_1 is a shortest non-zero vector in the lattice spanned by $b_1, b_2, \ldots, b_d$,

(c) The vectors $b_2 - \mu_{2,1}b_1, \ldots, b_d - \mu_{d,1}b_1$ are themselves HKZ-reduced,

where $\mu_{i,j} = (b_i, b_j^*)/\|b_j^*\|^2$ and b_i^* is the i-th vector of the Gram-Schmidt orthogonalisation of $(b_1, b_2, \ldots, b_d)$.

`HKZ(X): ModMatRngElt -> ModMatRngElt, AlgMatElt`

```
Proof : BoolElt                    Default: true
Unique: BoolElt                    Default: false
Prune : SeqEnum                    Default: [...,[1.0,...,1.0],...]
```

Given a matrix X over a subring of the real field, compute a matrix Y whose non-zero rows are an HKZ-reduced basis for the $\mathbb{Z}$-lattice spanned by the rows of X (which need not be $\mathbb{Z}$-linearly independent). The HKZ function returns an HKZ-reduced matrix Y whose rows span the same lattice ($\mathbb{Z}$-module) as that spanned by the rows of X, and a unimodular matrix T in the matrix ring over $\mathbb{Z}$ whose degree is the number of rows of X such that $TX = Y$. Although the implementation relies on floating-point arithmetic, the result can be guaranteed to be correct. By default, the `Proof` option is set to `true`, thus guaranteeing the output. The run-time might be improved by switching off this option. A given lattice may have many HKZ-reduced bases. If the `Unique` option is turned on, a uniquely determined HKZ-reduced basis is computed. This basis is chosen so that: for any $i > j$, we have $\mu_{i,j} \in [-1/2, 1/2)$; for any i, the first non-zero coordinate of the vector b_i^* is positive; and for any i, the vector b_i^* is the shortest among possible vectors for the lexicographical order. The implementation solves the Shortest Vector Problem for lattices of dimensions 1 to d, where d is the number of rows of the input matrix X. The latter instances of the Shortest Vector Problem are solved with the same enumeration algorithm as is used for computing the minimum of a lattice, which can be expressed as a search within a large tree. The i-th table of the `Prune` optional parameter is used to prune the i-dimensional enumeration. The default value is:

$[[1.0 : j \texttt{ in } [1..i]] : i \texttt{ in } [1..\texttt{NumberOfRows}(X)]]$.

See the introduction of Section *Minima and Element Enumeration* for more details on the `Prune` option.

`HKZGram(F): ModMatRngElt -> ModMatRngElt, AlgMatElt`

```
Proof: BoolElt                     Default: true
Prune: SeqEnum                     Default: [...,[1.0,...,1.0],...]
```

Given a symmetric positive semidefinite matrix F over a subring of the real field so that $F = XX^{tr}$ for some matrix X over the real field, compute a matrix G which is the Gram matrix corresponding to an HKZ-reduced form of the matrix X. The function returns the HKZ-reduced Gram matrix G of F, and a unimodular matrix T in the matrix ring over $\mathbb{Z}$ whose degree is the number of rows of F which gives the corresponding transformation: $G = TFT^{tr}$.

`HKZ(L): Lat -> Lat, AlgMatElt`

```
Proof: BoolElt                     Default: true
Prune: SeqEnum                     Default: [...,[1.0,...,1.0],...]
```

Given a lattice L with basis matrix B, return a new lattice L' with basis matrix B' and a transformation matrix T so that L' is equal to L but B' is HKZ-reduced and $B' = TB$.

`SetVerbose("HKZ", v): MonStgElt, RngIntElt`

(Procedure.) Set the verbose printing level for the HKZ algorithm to be v. Currently the legal values for v are `true`, `false`, 0, and 1 (`false` is the same as 0, and `true` is the same as 1). More information on the progress of the computation can be obtained by setting the `"Enum"` verbose on.

`GaussReduce(X): ModMatRngElt -> ModMatRngElt, AlgMatElt`

`GaussReduceGram(F): ModMatRngElt -> ModMatRngElt, AlgMatElt`

`GaussReduce(L): Lat -> Lat, AlgMatElt`

Restrictions of the `HKZ` functions to lattices of rank 2.

Example: HKZ Reduction (ex-421d95)

HKZ-reduced bases are much harder to compute than LLL-reduced bases, but provide a significantly more powerful representation of the spanned lattice. For example, computing all short lattice vectors is more efficient if one starts from an HKZ-reduced basis.

```
> d:=60;
> B:=RMatrixSpace(IntegerRing(), d, d)!0;
> for i,j in [1..d] do B[i][j]:=Random(100*d); end for;
> time C1 := LLL(B);
Time: 0.020
> time C2 := HKZ(B);
Time: 1.380
> m := Norm(C2[1]);
> time _:=ShortVectors(Lattice(C1), 11/10*m);
Time: 1.750
> time _:=ShortVectors(Lattice(C2), 11/10*m);
Time: 0.850
> time _:=ShortVectors(Lattice(C1), 3/2*m);
Time: 73.800
> time _:=ShortVectors(Lattice(C2), 3/2*m);
Time: 32.220
```

31.7.5 BKZ Reduction

The notion of block Korkine-Zolotareff reduction (BKZ) allows one to interpolate between LLL and HKZ. In particular, one gives a size parameter n, and at step i the HKZ-reduction of the projected sublattice on the vectors $[i..(i+n-1)]$ is computed. In practice, one achieves most of the possible reduction from (say) taking $n = 20$.

`BKZ(M, n): Mtrx, RngIntElt -> Mtrx, AlgMatElt, RngIntElt`

```
Delta: FldReElt                    Default: 0.75
```

Given a basis matrix M and a parameter n, return a BKZ-reduction.

`BKZ(L, n): Lat, RngIntElt -> Lat, AlgMatElt`

As above, but with a lattice (over the integers, rationals, or reals) as input.

Example: BKZ Reduction of a Random Basis (ex-13f155)

We create a 40-dimensional lattice, then take a random transformation of it. Then `LLL` is applied, but this does not give a great basis (at least with the chosen `Delta` value). Finally `BKZ` is used to make the basis a bit better.

```
> L := Lattice(LatticeDatabase(),"HZ40");
> L := LLL(L : Delta:=0.999);
> Max(Diagonal(GramMatrix(L)));
8
> R := RandomSLnZ(40,80,320);
> M := R*GramMatrix(L)*Transpose(R);
> A := LLLGram(M : Delta:=0.75);
> Max(Diagonal(A));
13
> LAT := BKZ(LatticeWithGram(A),20);
> Max(Diagonal(GramMatrix(LAT)));
6
```

31.7.6 Minkowski Reduction

Given a lattice L, an ordered set of d linearly independent lattice vectors $b_1, b_2, \ldots, b_d \in L$ is Minkowski-reduced if for all $1 \leq i \leq d$, b_i has minimal norm among all lattice vectors b_i such that $b_1, ..., b_i$ can be extended to a basis of L.

`MinkowskiReduction(L): Lat -> Lat, AlgMatElt`

`MinkowskiGramReduction(M): Mtrx -> Mtrx, Mtrx`

```
Canonical: BoolElt                    Default: false
```

Returns a lattice L' isometric to L but with a Minkowski-reduced basis, together with the corresponding transformation matrix T such that $M' = TMT^{tr}$, where M and M' are the Gram matrices of L and L' respectively.

If `Canonical` is `true`, returns a canonical representative L' so that L_1 and L_2 are isometric if and only if their canonical Minkowski-reduced representatives are equal. This is only implemented for lattices of rank at most 4.

31.7.7 Greedy Reduction

The Gauss reduction algorithm for binary lattices gives rise to a greedy algorithm, as described in [NS09c]. Given a lattice L, an ordered set of d linearly independent lattice vectors $b_1, b_2, \ldots, b_d \in L$ is greedy-reduced if for all $2 \leq i \leq d$ and for all $x_1, x_2, \ldots, x_{i-1} \in \mathbb{Z}$, we have $|b_i| \leq |b_i + x_1 b_1 + \ldots + x_{i-1} b_{i-1}|$.

Clearly, any Minkowski-reduced basis is greedy-reduced. In fact, for lattices of rank at most 4, these notions are equivalent. But for larger rank they may considerably differ.

`GreedyReduction(L): Lat -> Lat, AlgMatElt`

```
Al: MonStgElt                    Default: "Nguyen-Stehle"
```

Returns a lattice L' isometric to L but with a greedy-reduced basis, together with the corresponding transformation matrix T such that $M' = TMT^{tr}$, where M and M' are the Gram matrices of L and L' respectively.

The parameter `Al` specifies which algorithm to use for reduction. The default algorithm is the greedy Gauss reduction, described in [NS09c]. Another option is to set `Al := "Kohel"`, for a variant based on methods implemented by Kohel.

`GreedyOrbit(M): AlgMatElt -> Set[AlgMatElt]`

If M is a symmetric matrix on the boundary of a Minkowski domain, returns its orbit using a greedy algorithm.

31.7.8 Recovering a Short Basis from Short Lattice Vectors

`ReconstructLatticeBasis(S, B): ModMatRngElt, ModMatRngElt -> ModMatRngEltLat`

Given a basis S of a finite index sublattice of the lattice L spanned by the rows of the integral matrix B, return a matrix C whose rows span L and are not much longer than those of S. Specifically, the algorithm described in Lemma 7.1 of [MG02] is implemented, and the rows of the output matrix satisfy the following properties:

(a) For any i, $\|c_i\| \leq i^{1/2}\|s_i\|$,

(b) For any i, $\|c_i^*\| \leq \|s_i^*\|$,

where c_i^* (respectively s_i^*) denotes the i-th vector of the Gram-Schmidt orthogonalisation of $(c_1, c_2, \ldots, c_d)$ (respectively $(s_1, s_2, \ldots, s_d)$) .

31.8 Minima and Element Enumeration

The functions in this section are all based on one algorithm which enumerates all vectors of a lattice in a specified hyperball [FP83, Kan83, SE94]. As the underlying computational problems (the Shortest and Closest Lattice Vector Problems) are hard [Ajt98, vEB81], the general application of this algorithm is restricted to lattices of moderate dimension (up to 50 or 60). However, some tasks like finding a couple of short vectors or finding the minimum of a lattice without symmetry may still be feasible in higher dimensions. The function `EnumerationCost` provides an estimate of the cost of running the enumeration

algorithm on a specified input. This allows to know beforehand if the computation is likely to terminate within a reasonable amount of time.

For each function which enumerates short vectors of a lattice L, there is a corresponding function which enumerates vectors of L which are close to a given vector w (which usually lies outside L). Note that if one wishes to enumerate short vectors of the coset $L + w$ of the lattice L, where w is any vector of degree compatible with L, one can simply enumerate vectors $v \in L$ close to $-w$ and then just take the vectors $v + w$ for each v as the short vectors of the coset.

The enumeration routine underlying all the functions described below relies on floating-point approximations. However, it can be run in a rigorous way in some cases, see [PS08]. By default, the outputs of the functions `Minimum`, `PackingRadius`, `HermiteNumber`, `CentreDensity`, `Density`, `KissingNumber`, `ShortestVectors`, `ShortestVectorsMatrix`, `ShortVectors`, `ShortVectorsMatrix`, and `ThetaSeries` are guaranteed to be correct. This correctness guarantee can be turned off by setting the optional parameter `Proof` to `false`. However, this comes virtually for free, the additional computations being most often negligible. In the present version of Magma, the outputs of all other functions are only likely to be correct.

The enumeration algorithm from [FP83, Kan83, SE94] can be interpreted as a search within a large tree. This can be extremely time-consuming. Schnorr, Euchner, and Hörner [SE94, SH95] introduced techniques to prune the latter tree: the correct output might be missed, but the execution of the algorithm is likely to terminate faster. A new pruning strategy is available for the functions `Minimum`, `PackingRadius`, `HermiteNumber`, `CentreDensity`, `Density`, `KissingNumber`, `ShortestVectors`, `ShortestVectorsMatrix`, `ShortVectors`, `ShortVectorsMatrix`, and `ThetaSeries`. Naturally, if the pruning strategy is used, the result cannot be guaranteed to be correct anymore. Let $(b_1, b_2, \ldots, b_d)$ be a basis of a lattice L, let $(b_1^*, b_2^*, \ldots, b_d^*)$ denote its Gram-Schmidt orthogonalisation, and let $\mu_{i,j} = (b_i, b_j^*)/\|b_j^*\|^2$ for $i \geq j$. Suppose we are interested in finding all vectors in L of norm $\leq u$. The enumeration algorithm considers the equations: $\| \sum_{j=i}^{d} \left(\sum_{k=j}^{d} \mu_{k,j} x_k \right) b_j^* \|^2 \leq u$, for $i = d, d-1, \ldots, 1$,

where the x_k's are integers. If the `Prune` optional parameter is set to $[p_1, \ldots, p_d]$, then the equations above will be replaced by:

$\| \sum_{j=i}^{d} \left(\sum_{k=j}^{d} \mu_{k,j} x_k \right) b_j^* \|^2 \leq p_i u$, for $i = d, d-1, \ldots, 1$.

The p_i's must belong to the interval $[0, 1]$. For a given input $((b_1, \ldots, b_d), u)$ to the enumeration procedure, it is possible to heuristically estimate both the running-time gain and the probability of missing a solution. See Subsection *Lattice Enumeration Utilities* for more details.

31.8.1 Minimum, Density, and Kissing Number

The functions in this subsection compute invariants of a lattice which are all related to its minimum. See [JC98] for background about minimum, density, and kissing numbers.

`Minimum(L): Lat -> RngElt`

`Min(L): Lat -> RngElt`

```
Proof: BoolElt                  Default: true
Prune: SeqEnum                  Default: [1.0,...,1.0]
```

Return the minimum of the lattice L, i.e., the minimal norm of a non-zero vector in the lattice. Note that this is in general a hard problem and may be very time consuming. See also the attributes section below for how to assert the minimum of a lattice.

`PackingRadius(L): Lat -> FldReElt`

```
Proof: BoolElt                  Default: true
Prune: SeqEnum                  Default: [1.0,...,1.0]
```

The packing radius is half the square root of the minimum of the non-zero lattice L.

`HermiteConstant(n): RngIntElt -> RngElt`

Return the n-th Hermite constant raised to the power of n. The exact value is provided if $n \leq 8$ or $n = 24$, and otherwise an upper bound is returned. The n-th Hermite constant is defined as the maximum of $\texttt{Min}(L)/\texttt{Determinant}(L)^{1/n}$ over all n-dimensional lattices L, and this intrinsic returns the nth power of this.

`HermiteNumber(L): Lat -> FldReElt`

```
Proof: BoolElt                  Default: true
Prune: SeqEnum                  Default: [1.0,...,1.0]
```

Return the Hermite number, $\texttt{Min}(L)/\texttt{Determinant}(L)^{1/\texttt{Dimension}(L)}$, of the non-zero lattice L.

`CentreDensity(L): Lat -> FldReElt`

`CenterDensity(L): Lat -> FldReElt`

`CentreDensity(L, K): Lat, Fld -> FldElt`

`CenterDensity(L, K): Lat, Fld -> FldElt`

```
Proof: BoolElt                  Default: true
Prune: SeqEnum                  Default: [1.0,...,1.0]
```

The center density of the lattice L, as an element of the real field K. This is defined to be the square root of $(\texttt{Min}(L)/4)^{\texttt{Rank(L)}}/\texttt{Determinant(L)}$. The argument for the real field K may be omitted, in which case K is taken to be the current default real field.

The product of the centre density by the volume of a sphere of radius 1 in n-dimensional space gives the density of the lattice-centered sphere packing of L, called the density of the lattice.

`Density(L): Lat -> FldReElt`

`Density(L, K): Lat, Fld -> FldReElt`

```
Proof: BoolElt                  Default: true
Prune: SeqEnum                  Default: [1.0,...,1.0]
```

The density of the lattice L, as an element of the real field K, i.e., the density of the lattice-centered sphere packing. If the argument for the real field K is omitted, the field K is taken to be the default real field.

`KissingNumber(L): Lat -> RngElt`

```
Proof: BoolElt                  Default: true
Prune: SeqEnum                  Default: [1.0,...,1.0]
```

Return the kissing number of the lattice L, which equals the number of vectors of minimal non-zero norm (twice the length of the sequence returned by `ShortestVectors(L)` since that returns only normalized vectors). This is the maximum number of nonoverlapping spheres of diameter `Minimum(L)` with centres at lattice points which touch a fixed sphere of the same diameter with centre at a lattice point.

Example: Invariants of the Leech Lattice (ex-0f67ef)

We create the Leech lattice Λ_{24} and compute its minimum (4), density, and kissing number (196560). Note that the computation of the minimum is very fast since the lattice is even and at least one basis element has norm 4; one has only to prove that there are no vectors of norm 2.

```
> L := Lattice("Lambda", 24);
> IsEven(L), Norm(L.2);
true 4
> time Minimum(L);
4
```

(continues on next page)

(continued from previous page)

```
Time: 0.020
> Density(L);
0.0019295743094039230479033455636 9
> time KissingNumber(L);
196560
Time: 0.180
```

31.8.2 Shortest and Closest Vectors

ShortestVectors(L): Lat -> [LatElt]

Max :	RngIntElt	Default: Infinity()
Proof:	BoolElt	Default: true
Prune:	SeqEnum	Default: [1.0,...,1.0]

Return the shortest non-zero vectors of the lattice L, i.e., the vectors $v \in L$ such that (v, v) is minimal, as a sorted sequence Q. The vectors are computed up to sign (so only one of v and $-v$ appears in Q) and normalized so that the first non-zero entry in each vector is positive, and Q is sorted with respect to lexicographic order. By default, all the (normalized) vectors are computed; the optional parameter Max allows the user to specify the maximal number of computed vectors. Note that unless the minimum of the lattice is already known, it has to be computed by this function, which may be very time consuming.

ShortestVectorsMatrix(L): Lat -> ModMatRngElt

Max :	RngIntElt	Default: Infinity()
Proof:	BoolElt	Default: true
Prune:	SeqEnum	Default: [1.0,...,1.0]

Return the shortest non-zero vectors of the lattice L as the rows of a matrix. This is more efficient or convenient for some applications than forming the sequence of vectors. Note that the matrix will lie in the appropriate matrix space and the inner product for L will not be related to the rows of the matrix (which will lie in the appropriate R-space). By default, all the (normalized) vectors are computed; the optional parameter Max allows the user to specify the maximal number of computed vectors.

ShortestVector(L): Lat -> LatElt

Return a single non-zero shortest vector of the lattice L. This is equivalent to calling ShortestVectors with the parameter Max := 1 and extracting the single vector.

ClosestVectors(L, w): Lat, ModTupRngElt -> [LatElt], RngElt

ClosestVectors(L, w): Lat, LatElt -> [LatElt], RngElt

Max:	RngIntElt	Default: Infinity()

Return the vectors of the lattice L which are closest to the vector w, together with the minimal squared distance d. w may be any element of a lattice of degree n or an R-space of degree n compatible with L, where n is the degree of L. The closest vectors are those vectors $v \in L$ such that the squared distance $(v - w, v - w)$ between v and w (and thus the distance between v and w) is minimal; this minimal squared distance is the second return value d. The vectors are returned as a sequence Q, sorted with respect to lexicographic order. Note that the closest vectors are *not* symmetrical with respect to sign (while the shortest vectors are) so the returned closest vectors are not normalized. By default, all the closest vectors are computed; the optional parameter Max allows the user to specify the maximal number of computed vectors.

ClosestVectorsMatrix(L, w): Lat, ModTupRngElt -> ModMatRngElt, RngElt

`ClosestVectorsMatrix(L, w): Lat, LatElt -> ModMatRngElt, RngElt`

```
Max: RngIntElt                    Default: Infinity()
```

Return the vectors of the lattice L which are closest to the vector w as a matrix, together with the squared distance d. w may be any element of a lattice of degree n or an R-space of degree n compatible with L, where n is the degree of L. Note that the matrix will lie in the appropriate matrix space and the inner product for L will not be related to the rows of the matrix (which will lie in the appropriate R-space). The vectors are returned as sequence Q, sorted with respect to lexicographic order. Note that the closest vectors are *not* symmetrical with respect to sign (while the shortest vectors are) so the returned closest vectors are not normalized. By default, all the closest vectors are computed; the optional parameter `Max` allows the user to specify the maximal number of computed vectors.

`ClosestVector(L, w): Lat, ModTupRngElt -> LatElt`

`ClosestVector(L, w): Lat, LatElt -> LatElt`

Return a single vector of the lattice L which is closest to the vector w. This is equivalent to calling `ClosestVectors` with the parameter `Max := 1` and extracting the single vector.

Example: Shortest Vectors of the E8 Lattice (ex-43a78a)

We create the Gosset lattice $L = E_8$ and find the shortest vectors of L. There are 120 normalized vectors so the kissing number is 240, and the minimum is 2.

```
> L := Lattice("E", 8);
> S := ShortestVectors(L);
> #S;
120
> KissingNumber(L);
240
> { Norm(v): v in S };
{ 2 }
> Minimum(L);
2
```

We note that the rank of the space generated by the shortest vectors is 8 so that the successive minima of L are $[2, 2, 2, 2, 2, 2, 2, 2]$ (see the function `SuccessiveMinima` below).

```
> Rank(ShortestVectorsMatrix(L));
8
```

We next find the vectors in L which are closest to a certain vector in the Q-span of L. The vector is an actual hole of L and the square of its distance from L is 8/9.

```
> w := RSpace(RationalField(), 8) !
>   [ -1/6, 1/6, -1/2, -1/6, 1/6, -1/2, 1/6, -1/2 ];
> C, d := ClosestVectors(L, w);
> C;
[
    (-1/2 -1/2 -1/2 -1/2  1/2 -1/2  1/2 -1/2),
    (-1/2  1/2 -1/2 -1/2 -1/2 -1/2  1/2 -1/2),
    (-1/2  1/2 -1/2 -1/2  1/2 -1/2 -1/2 -1/2),
    (-1/2  1/2 -1/2  1/2  1/2 -1/2  1/2 -1/2),
    ( 1/2  1/2 -1/2 -1/2  1/2 -1/2  1/2 -1/2),
    ( 0  0 -1  0  0 -1  0  0),
    ( 0  0 -1  0  0  0  0 -1),
    ( 0  0  0  0  0 -1  0 -1),
    (0 0 0 0 0 0 0 0)
]
```

(continues on next page)

(continued from previous page)

```
> d;
8/9
> { Norm(v): v in C };
{ 0, 2 }
```

We verify that the squared distance of the vectors in C from w is $8/9$.

```
> { Norm(v - w): v in C };
{ 8/9 }
```

We finally notice that these closest vectors are in fact amongst the shortest vectors of the lattice (together with the zero vector).

```
> Set(C) subset (Set(S) join {-v: v in S} join { L!0 });
true
```

31.8.3 Short and Close Vectors

ShortVectors(L, u): Lat, RngElt -> [<LatElt, RngElt>]

ShortVectors(L, l, u): Lat, RngElt, RngElt -> [<LatElt, RngElt>]

```
Max  : RngIntElt                    Default: Infinity()
Proof: BoolElt                      Default: true
Prune: SeqEnum                      Default: [1.0,...,1.0]
```

Return the vectors of the lattice L with norm within the prescribed range, together with their norms, as a sorted sequence Q. The sequence Q contains tuples of the form $< v, r >$ where v is a vector and r its norm. Either one positive number u can be given, specifying the range $(0, u]$, or a pair l, u of positive numbers, specifying the range $[l, u]$. The vectors are computed up to sign (so only one of v and $-v$ appears in Q) and normalized so that the first non-zero entry in each vector is positive, and Q is sorted with respect to first the norms and then the lexicographic order for the vectors. By default, all the (normalized) vectors with norm in the prescribed range are computed. The optional parameter **Max** allows the user to specify the maximal number of computed vectors.

ShortVectorsMatrix(L, u): Lat, RngElt -> ModMatRngElt

ShortVectorsMatrix(L, l, u): Lat, RngElt, RngElt -> ModMatRngElt

```
Max  : RngIntElt                    Default: Infinity()
Proof: BoolElt                      Default: true
Prune: SeqEnum                      Default: [1.0,...,1.0]
```

This is very similar to **ShortVectors**, but returns the vectors of the lattice L with norm within the prescribed range as the rows of a matrix S rather than as a sequence of tuples. This is more efficient or convenient for some applications than forming the sequence. By default, all the (normalized) vectors with norm in the prescribed range are computed. The optional parameter **Max** allows the user to specify the maximal number of computed vectors. Note that the matrix will lie in the appropriate matrix space and the inner product for L will not be related to the rows of the matrix (which will lie in the appropriate R-space).

ShortVector(L, u): Lat, RngElt -> LatElt

ShortVector(L, l, u): Lat, RngElt, RngElt -> LatElt

Return a single non-zero vector of the lattice L with norm within the prescribed range, or the zero vector of L if there is no such vector. This is equivalent to calling one of the above functions with the parameter **Max** := 1 and extracting the single vector when it exists.

CloseVectors(L, w, u): Lat, ModTupRngElt, RngElt -> [<LatElt, RngElt>]

CloseVectors(L, w, l, u): Lat, ModTupRngElt, RngElt, RngElt -> [<LatElt, RngElt>]

`CloseVectors(L, w, u): Lat, LatElt, RngElt -> [ <LatElt, RngElt> ]`

`CloseVectors(L, w, l, u): Lat, LatElt, RngElt, RngElt -> [ <LatElt, RngElt> ]`

`Max: RngIntElt` `Default: Infinity()`

Return the vectors of the lattice L whose squared distance from the vector w is within the prescribed range, together with their squared distances, as a sorted sequence Q. The returned sequence Q contains tuples of the form $< v, d >$ where v is a vector from L and $d = (v - w, v - w)$ is its squared distance from w. w may be any element of a lattice of degree n or an R-space of degree n compatible with L, where n is the degree of L. Either one positive number u can be given, specifying the range $(0, u]$, or a pair l, u of positive numbers, specifying the range $[l, u]$. Note that close vectors are *not* symmetrical with respect to sign (while short vectors are) so the returned close vectors are not normalized. Q is sorted with respect to first the squared distances and then the lexicographic order for the vectors. By default, all the vectors with squared distance in the prescribed range are computed. The optional parameter `Max` allows the user to specify the maximal number of computed vectors.

Note that `CloseVectors` does not try to improve the given basis for enumeration purposes (for instance, with `LLL` and `EnumerationCost`). It always operates with respect to the `BasisMatrix` (and `InnerProductMatrix`) of the lattice, though internal manipulations are done to move the problem from the input ring(s) to the integers before an enumeration is done.

Also, if the target vector is in the lattice, one can gain a factor of 2 by transforming the problem to use `ShortVectors` (which identifies a vector with its additive inverse in the output).

`CloseVectorsMatrix(L, w, u): Lat, ModTupRngElt, RngElt -> ModMatRngElt`

`CloseVectorsMatrix(L, w, l, u): Lat, ModTupRngElt, RngElt, RngElt -> ModMatRngElt`

`CloseVectorsMatrix(L, w, u): Lat, LatElt, RngElt -> ModMatRngElt`

`CloseVectorsMatrix(L, l, u): Lat, LatElt, RngElt, RngElt -> ModMatRngElt`

`Max: RngIntElt` `Default: Infinity()`

This is very similar to `CloseVectors`, but returns the vectors of the lattice L whose squared distance from the vector w is within the prescribed range as the rows of a matrix C rather than as a sequence of tuples. This is more efficient or convenient for some applications than forming the sequence. w may be any element of a lattice of degree n or an R-space of degree n compatible with L, where n is the degree of L. By default, all the close vectors are computed. The optional parameter `Max` allows the user to specify the maximal number of computed vectors. Note that the matrix will lie in the appropriate matrix space and the inner product for L will not be related to the rows of the matrix (which will lie in the appropriate R-space).

`CloseVector(L, w, u): Lat, ModTupRngElt, RngElt -> LatElt`

`CloseVector(L, w, l, u): Lat, ModTupRngElt, RngElt, RngElt -> LatElt`

`CloseVector(L, w, u): Lat, LatElt, RngElt -> LatElt`

`CloseVector(L, w, l, u): Lat, LatElt, RngElt, RngElt -> LatElt`

Return a single non-zero vector of the lattice L whose squared distance from the vector w is within the prescribed range, or the zero vector of L if there is no such vector. This is equivalent to calling one of the above functions with the parameter `Max := 1` and extracting the single vector when it exists.

Example: Solving a Knapsack Problem (ex-76fabb)

Let $Q = [a_1, \ldots, a_n]$ be a sequence of (not necessarily distinct) positive integers and let s be a positive integer. We wish to find all solutions to the equation $\sum_{i=1}^{n} x_i a_i = s$ with $x_i \in \{0, 1\}$. This is known as the *Knapsack* problem. The following lattice-based solution is due to Schnorr and Euchner (op. cit., at the beginning of this chapter). To solve the problem, we create the lattice L

of rank $n+1$ and degree $n+2$ with the following basis:

$$\begin{aligned} b_1 &= (2, 0, \ldots, 0, na_1, 0) \\ b_2 &= (0, 2, \ldots, 0, na_2, 0) \\ &\vdots \\ b_n &= (0, 0, \ldots, 2, na_n, 0) \\ b_{n+1} &= (1, 1, \ldots, 1, ns, 1). \end{aligned}$$

Then every vector $v = (v_1, \ldots, v_{n+2}) \in L$ such that the norm of v is $n+1$ and

$$v_1, \ldots, v_n, v_{n+2} \in \{\pm 1\}, v_{n+1} = 0,$$

yields the solution $x_i = |v_i - v_{n+2}|/2$ for $i = 1, \ldots, n$ to the original equation.

We first write a function `KnapsackLattice` which, given the sequence Q and sum s, creates a matrix X representing the above basis and returns the lattice generated by the rows of X. Note that the `Lattice` creation function will automatically LLL-reduce the matrix X as it creates the lattice.

```
> function KnapsackLattice(Q, s)
>     n := #Q;
>     X := RMatrixSpace(IntegerRing(), n + 1, n + 2) ! 0;
>     for i := 1 to n do
>         X[i][i] := 2;
>         X[i][n + 1] := n * Q[i];
>         X[n + 1][i] := 1;
>     end for;
>     X[n + 1][n + 1] := n * s;
>     X[n + 1][n + 2] := 1;
>     return Lattice(X);
> end function;
```

We next write a function `Solutions` which uses the function `ShortVectors` to enumerate all vectors of the lattice L having norm exactly $n+1$ and thus to find all solutions to the Knapsack problem associated with L. (Note that the minimum of the lattice may be less than $n+1$.) The function returns each solution as a sequence of indices for Q.

```
> function KnapsackSolutions(L)
>     n := Rank(L) - 1;
>     M := n + 1;
>     S := ShortVectors(L, M, M);
>     return [
>         [i: i in [1 .. n] | v[i] ne v[n + 2]]: t in S |
>             forall{i: i in [1 .. n] cat [n + 2] | Abs(v[i]) eq 1} and
>                 v[n + 1] eq 0 where v is t[1]
>     ];
> end function;
```

We now apply our functions to a sequence Q of 12 integers each less than 1000 and the sum 2676. There are actually 4 solutions. We verify that each gives the original sum.

```
> Q := [ 52, 218, 755, 221, 574, 593, 172, 771, 183, 810, 437, 137 ];
> s := 2676;
> L := KnapsackLattice(Q, s);
> L;
Lattice of rank 13 and degree 14
Determinant: 1846735827632128
Basis:
```

(continues on next page)

(continued from previous page)

```
( 0  0  0  0 -2  0  0  0  0  0  2  2  0  0)
( 1  1 -1 -1 -1  1 -1 -1 -1  1  1  1  0  1)
( 1 -1 -1 -1 -1  1  1 -1  1  1  1 -1  0  1)
( 1  1  1  1 -3  1  1 -1 -1  1 -1 -1  0  1)
( 1 -1 -3  1  1 -1 -1  1 -1  1  1  1  0  1)
( 2  2  0  0  0 -2  2  2 -2  0 -2  0  0  0)
( 3 -1  1  1 -1  1 -1 -1 -1 -1  1  1  0  1)
( 1 -1  1  1  1  1  3 -1 -1 -1 -1  1  0 -1)
(-1 -1 -1 -1  1  1  1  1 -1 -1 -1  1  0  1)
( 2  0  0  2  0  2  0  0  0  0 -2  0  0 -2)
(-1 -1 -1  1  1 -1 -1  1 -1  1  1 -3  0 -1)
(-1 -1  1 -1 -1  1 -1 -1 -1  3 -1  1  0 -3)
( 0 -2  0  0  2  0 -2  0 -2  0  0  0 12  0)
> S := KnapsackSolutions(L);
> S;
[
    [ 2, 3, 4, 5, 8, 12 ],
    [ 3, 4, 5, 7, 8, 9 ],
    [ 3, 4, 7, 8, 9, 11, 12 ],
    [ 1, 2, 3, 4, 9, 10, 11 ]
]
> [&+[Q[i]: i in s]: s in S];
[ 2676, 2676, 2676, 2676 ]
```

Finally, we apply our method to a larger example. We let Q be a sequence consisting of 50 random integers in the range $[1, 2^{1000}]$. We let I be a random subset of $\{1 \dots 50\}$ and let s be the sum of the elements of Q indexed by I. We then solve the Knapsack problem with input (Q, s) and this time obtain I as the only answer.

```
> b := 1000;
> n := 50;
> SetSeed(1);
> Q := [Random(1, 2^b): i in [1 .. n]];
> I := { };
> while #I lt n div 2 do
>     Include(~I, Random(1, n));
> end while;
> I := Sort(Setseq(I)); I;
[ 1, 3, 4, 7, 10, 11, 13, 14, 18, 20, 22, 23, 26, 28, 29, 34, 35, 37, 40, 41,
42, 45, 48, 49, 50 ]
> s := &+[Q[i]: i in I]; Ilog2(s);
1003
> time L := KnapsackLattice(Q, s);
Time: 0.570
> [Ilog2(Norm(b)): b in Basis(L)];
[ 5, 46, 46, 45, 47, 47, 46, 46, 46, 46, 46, 46, 46, 45, 47, 46, 46, 46, 46,
47,46, 46, 45, 46, 46, 46, 46, 46, 46, 46, 46, 46, 46, 46, 46, 46, 45, 46, 46,
45, 45, 46, 46, 46, 46, 46, 45, 46, 46, 46, 46, 46 ]
> time KnapsackSolutions(L);
[
    [ 1, 3, 4, 7, 10, 11, 13, 14, 18, 20, 22, 23, 26, 28, 29, 34, 35, 37, 40,
    41, 42, 45, 48, 49, 50 ]
]
Time: 0.040
```

Example: Splitting Representations with Short Vectors (ex-92d5fa)

In this example we demonstrate how short vectors of lattices can be used to split homogeneous components of integral representations. We first define an integral matrix group of degree 8. The group is isomorphic to A_5 and the representation contains the 4-dimensional irreducible representation of A_5 with multiplicity 2.

```
> G := MatrixGroup< 8, Integers() |
>   [ -673,  -291,  -225,  -316,   250,   -32,    70,  -100,
>      252,   274,   349,   272,  -156,   -94,  -296,   218,
>     2532,  1159,   609,  5164, -1450,  -181,   188,   742,
>     -551,   163,   629, -1763,   285,  -162,  -873,   219,
>    -2701,  -492,   411, -3182,  1062,  -397, -1195,   151,
>    -5018, -1112,  1044,-12958,  2898,  -153, -2870,  -454,
>     2581,    90, -1490,  8197, -1553,   261,  2556,  -149,
>    -3495, -2776, -3218, -2776,  1773,   652,  2459, -1910],
>   [ 2615,  1314,  1633,   400,  -950, -1000, -2480,  1049,
>      161,   159,   347,  -657,     2,  -385,  -889,   230,
>    -2445, -1062, -1147,   269,   744,  1075,  2441,  -795,
>     1591,   925,  1454, -1851,  -350, -1525, -3498,   982,
>    10655,  5587,  7476, -1751, -3514, -5575,-13389,  4873,
>     6271,  3253,  4653, -6126, -1390, -5274,-11904,  3219,
>    -3058, -1749, -2860,  5627,   392,  3730,  8322, -2009,
>     4875,  1851,  1170,  5989, -2239,   625,  1031,   692] >;
> Order(G);
60
```

Since the group is small enough we can generate elements in the endomorphism ring by averaging over the group elements.

```
> M := MatrixRing(Integers(), 8);
> e := [ &+[ M!g * MatrixUnit(M, i, i) * M!(g^-1) : g in G ] : i in [1..4] ];
> E := sub<M | e>;
> Dimension(E);
4
```

We now transform E into a lattice of dimension 4 and degree 64 and rescale the basis vectors so that they have lengths of the same order of magnitude.

```
> L := Lattice(64, &cat[ Eltseq(b) : b in Basis(E) ]);
> LL := sub<L | [ Round( Norm(L.4)/Norm(L.i) ) * L.i : i in [1..4] ]>;
> Minimum(LL);
910870284600
> SV := ShortVectors(LL, 100*Minimum(LL));
> #SV;
46
> Sing := [ X : v in SV | Determinant(X) eq 0 where X is M!Eltseq(v[1]) ];
> #Sing;
3
```

We thus have found three singular elements amongst the 46 shortest vectors and use the kernel of the first of these to get the representation on a subspace of dimension 4.

```
> ker := LLL( KernelMatrix(Sing[1]) );
> ker;
[ 10   0  -9   5  -2   0   5  -4]
[ -3  -4   1 -16   3   5   4  -3]
[ -8 -11 -10  -1   4   0   6  -5]
[-13   3   4  10   1   5   8   2]
> H := MatrixGroup<4, Integers() | [Solution(ker, ker*g) : g in Generators(G)]>;
```

(continues on next page)

(continued from previous page)

```
> H;
MatrixGroup(4, Integer Ring)
Generators:
    [ 1  0  0  0]
    [-3 -2 -4 -5]
    [-2 -3 -3 -4]
    [ 2  3  4  5]

    [-1 -1 -1 -1]
    [ 2  1  2  2]
    [-4 -2 -1 -2]
    [ 3  2  1  2]
> #H;
60
```

31.8.4 Short and Close Vector Processes

ShortVectorsProcess(L, u): Lat, RngElt -> LatEnumProc

ShortVectorsProcess(L, l, u): Lat, RngElt, RngElt -> LatEnumProc

Given a lattice L and a range, create a corresponding short vectors lattice enumeration process P. This process provides the environment for enumerating each vector $v \in L$ with norm within the range. Either a positive number u can be given, specifying the range $(0, u]$, or a pair l, u of positive numbers, specifying the range $[l, u]$. Successive calls to `NextVector` (see below) will result in the enumeration of the vectors.

CloseVectorsProcess(L, w, u): Lat, ModTupRngElt, RngElt -> LatEnumProc

CloseVectorsProcess(L, w, l, u): Lat, ModTupRngElt, RngElt, RngElt -> LatEnumProc

CloseVectorsProcess(L, w, u): Lat, LatElt, RngElt -> LatEnumProc

CloseVectorsProcess(L, w, l, u): Lat, LatElt, RngElt, RngElt -> LatEnumProc

Given a lattice L, a vector w, and a range, create a corresponding close vectors lattice enumeration process P. This process provides the environment for enumerating each vector $v \in L$ such its squared distance $(v - w, v - w)$ from w is within the range. w may be any element of a lattice of degree n or an R-space of degree n compatible with L, where n is the degree of L. Either a positive number u can be given, specifying the range $(0, u]$, or a pair l, u of positive numbers, specifying the range $[l, u]$. Successive calls to `NextVector` (see below) will result in the enumeration of the vectors.

NextVector(P): LatEnumProc -> LatElt, RngElt

Given a lattice enumeration process P as created by `ShortVectorsProcess` or `CloseVectorsProcess`, return the next element found in the enumeration. If the process is for short vectors, the next short vector of the specified region of the lattice is returned together with its norm, or the zero vector together with -1, indicating that the enumeration process has been completed. The vectors are computed up to sign (so that only one of v and $-v$ will be enumerated) and normalized so that the first non-zero entry in each vector is positive. If the process is for the vectors close to w, the next close vector v whose squared distance $d = (v - w, v - w)$ from w is in the specified range is returned together with d, or the zero vector together with -1, indicating that the enumeration process has been completed. The close vectors are *not* symmetrical with respect to sign (while short vectors are) so the returned close vectors are not normalized. Note also that to test for completion the second return value should be tested for equality with -1 (or, preferably, the next function `IsEmpty` should be used) since the zero vector could be a valid close vector. Note that the order of the vectors returned by this function in each case is arbitrary, unlike the previous functions where the resulting sequence or matrix is sorted.

IsEmpty(P): LatEnumProc -> BoolElt

Given a lattice enumeration process P, return whether the process P has found all short or close vectors.

31.8.5 Successive Minima and Theta Series

`SuccessiveMinima(L): Lat -> [ RngElt ], [ LatElt ]`

`SuccessiveMinima(L, k): Lat, RngIntElt -> [ RngElt ], [ LatElt ]`

Return the first k successive minima of lattice L, or all of the m successive minima of L if k is omitted, where m is the rank of L. The first k successive minima $M_1, \ldots, M_k$ of a lattice L are defined by the property that $M_1, \ldots, M_k$ are minimal such that there exist linearly independent vectors $l_1, \ldots, l_k$ in L with $(l_i, l_i) = M_i$ for $1 \leq i \leq k$. The function returns a sequence containing the minima M_i and a sequence containing the vectors l_i. The lattice L must be an exact lattice (over $\mathbb{Z}$ or $\mathbb{Q}$). Note that the minima are unique but the vectors are not.

This uses `ShortVectors` and then tests for independence, and thus can be quite expensive. In particular, for skewed lattices in low dimensions it is often not the right tool, as many inutile short vectors will be found. In many cases, the diagonal entries from the Gram matrix of a suitable reduction are sufficiently good approximations. An example where the problem can be seen to occur: a 20-dimensional lattice with a 5-dimensional subspace generated by vectors of norm 10^5, with the shortest vector not in this subspace having around norm 10^{90}. To rigorously determine the 6th minima by direct vector enumeration is hopeless, and to use trickier methods, such as cosets of the subspace, would involve around $(10^5)^5$ sublattices.

Example: Successive Minima of a Lattice `(ex-c16c45)`

The successive minima count only linearly independent vectors, so a short vector that is a multiple of an earlier minimum vector does not contribute a new minimum. Consider the rank 2 lattice whose Gram matrix is diagonal with entries 1/49 and 5/49.

```
> L := LatticeWithGram(Matrix([[1/49, 0], [0, 5/49]]));
> m, vecs := SuccessiveMinima(L);
> m;
[ 1/49, 5/49 ]
```

The lattice does contain a vector of norm 4/49, but it is twice the first minimum vector and so cannot serve as a second, linearly independent, minimum.

```
> S := ShortVectors(L, 5/49);
> S;
[
<(1 0), 1/49>,
<(2 0), 4/49>,
<(0 1), 5/49>
]
> S[2][1] eq 2*vecs[1];
true
```

The smallest norm of a vector independent of the first minimum vector is 5/49, so the second successive minimum is 5/49, not 4/49.

`ThetaSeries(L, n): Lat, RngIntElt -> RngSerElt`

```
Proof: BoolElt                  Default: true
Prune: SeqEnum                  Default: [1.0,...,1.0]
```

Given an integral lattice L and a small positive integer n, return the Theta series $\Theta_L(q)$ of L as a formal power series in q to precision n (i.e., up to and including the coefficient of q^n). The coefficient of q^k in $\Theta_L(q)$ is defined to be the number of vectors of norm k in L. Note that this function needs to enumerate all vectors of L having norm up to and including n, so its application is restricted to lattices where the number of these vectors is reasonably small. The lattice L must be an exact lattice (over $\mathbb{Z}$ or $\mathbb{Q}$). Note that the angle bracket notation should be used to assign a name to the indeterminate for the returned Theta series (e.g., `T<q> := ThetaSeries(L);`).

Example: Computing a Theta Series (ex-e2016c)

We show how a lattice enumeration process can be used to compute the Theta series of a lattice. We write a simple function `Theta` which takes a lattice L and precision n and returns the Theta series of L up to the term q^n just as in the function `ThetaSeries`. The function assumes that the lattice L is integral. We simply loop over the non-zero vectors of norm up to n and count the number of vectors for each norm.

```
> function Theta(L, n)
>     Z := IntegerRing();
>     P := ShortVectorsProcess(L, n);
>     C := [1] cat [0: i in [1 .. n]];
>     while not IsEmpty(P) do
>         v, norm := NextVector(P);
>         C[Z!norm + 1] +:= 2;
>     end while;
>     return PowerSeriesRing(IntegerRing()) ! C;
> end function;
```

We now compute the Theta series up to norm 10 of the Gosset lattice E_8 using the function `Theta`. We compare this Magma-language version with the builtin function `ThetaSeries` (which is much faster of course because of the lack of interpreter overhead etc.).

```
> L := Lattice("E", 8);
> time T<q> := Theta(L, 10);
Time: 0.050
> T;
1 + 240*q^2 + 2160*q^4 + 6720*q^6 + 17520*q^8 + 30240*q^10
> time TT<r> := ThetaSeries(L, 10);
Time: 0.000
> TT;
1 + 240*q^2 + 2160*q^4 + 6720*q^6 + 17520*q^8 + 30240*q^10 + O(q^11)
```

`ThetaSeriesIntegral(L, n): Lat, RngIntElt -> RngSerElt`

```
Proof: BoolElt                  Default: true
Prune: SeqEnum                  Default: [1.0,...,1.0]
```

Restriction of `ThetaSeries` to integral lattices.

31.8.6 Lattice Enumeration Utilities

`SetVerbose("Enum", v): MonStgElt, RngIntElt`

(Procedure.) Set the verbose printing level for the lattice enumeration algorithm to be v. Currently the legal values for v are `true`, `false`, 0, and 1 (`false` is the same as 0, and `true` is the same as 1). When the verbose level is non-zero, some information about the current status of the enumeration algorithm is printed out every 15 seconds. This concerns the functions `Minimum`, `CentreDensity`, `CenterDensity`, `Density`, `KissingNumber`, `ShortestVectors`, `ShortestVectorsMatrix`, `ShortVectors`, `ShortVectorsMatrix`, and `ThetaSeries`.

`EnumerationCost(L): Lat -> FldReElt`

`EnumerationCost(L, u): Lat, FldReElt -> FldReElt`

```
Prune: SeqEnum                  Default: [1.0,...,1.0]
```

Estimate the number of nodes in the tree to be visited during the execution of the enumeration algorithm, for the lattice L and an hyperball of squared radius u. If u is not provided, an upper bound to the lattice minimum is used. Since the cost per tree node is relatively constant, this function can be used to obtain a heuristic estimate of the running-time of the enumeration algorithm.

Note that in some cases the enumeration may run faster than expected, because the tree to be visited may be shrunk during the execution, as described in [SE94]. Contrary to the enumeration itself, this function runs in time polynomial in the input bit-size (if the value of the `Prune` optional parameter is the default one). It relies on the Gaussian heuristic, which consists in estimating the number of integral points within an n-dimensional body by its volume (see [HS07] for more details).

If the `Prune` parameter is set to $[p_1, \ldots, p_d]$, then `EnumerationCost` estimates the cost of the tree pruned with the strategy described in the introduction of the present section, with pruning coefficients $p_1, \ldots, p_d$. Although the enumeration itself is likely to terminate faster, the estimation of the cost may be significantly more time-consuming.

The actual speed of enumeration depends on the dimension, and to a lesser extent on the shape of the lattice. The main task at each step is to calculate an inner product; if the dimension is large this will take longer, while if the lattice doesn't have too many short vectors, the tree search will not spend so much time with long(er) inner products. As a general guess, on a desktop machine 25 million vectors can be searched per second in dimension 40.

`EnumerationCostArray(L): Lat -> ModTupFldElt`

`EnumerationCostArray(L, u): Lat, FldReElt -> ModTupFldElt`

```
Prune: SeqEnum                    Default: [1.0,...,1.0]
```

Estimate the number of nodes in each layer of the tree to be visited during the execution of the enumeration algorithm, for the lattice L and a hyperball of squared radius u. If u is not provided, an upper bound to the lattice minimum is used.

Example: Estimating Enumeration Cost (ex-2bc3fa)

The algorithm that enumerates short lattice vectors may be very time-consuming. The `EnumerationCost` and `EnumerationCostArray` functions allow one to estimate the running-time on a given enumeration instantiation, before actually running the algorithm. If the running-time estimate is too high, then it is unlikely that the execution will terminate within a reasonable amount of time. A good strategy then consists in reducing the input lattice basis further. If the running-time estimate remains too high, then pruning the enumeration tree should be considered. Both the running-time gain and the "probability" of missing a solution can be estimated.

```
> B:=RMatrixSpace(IntegerRing(), 50, 51) ! 0;
> for i := 1 to 50 do B[i][1] := RandomBits(1000); end for;
> for i := 1 to 50 do B[i][i+1] := 1; end for;
> B := LLL(B);
> EnumerationCost (Lattice(B));
4.06E18
```

The value above is an estimate of the number of nodes in the enumeration tree that corresponds to the computation of the shortest non-zero vectors in the lattice spanned by the rows of B. This is much too high to have a chance to terminate. Reducing B further allows us to compute these shortest non-zero vectors.

```
> B:=LLL(B:Delta:=0.999);
> EnumerationCost(Lattice(B));
1.03E13
> B:=LLL(B:Delta:=0.999, DeepInsertions);
> EnumerationCost(Lattice(B));
7.43E7
> time _:=ShortestVectors(Lattice(B));
3.660
```

In larger dimensions, reducing the lattice further may not prove sufficient to make the enumeration reasonably tractable. Then pruning the enumeration tree can be considered.

```
>  B:=RMatrixSpace(IntegerRing(), 65, 66) ! 0;
>  for i := 1 to 65 do B[i][1] := RandomBits(1000); end for;
>  for i := 1 to 65 do B[i][i+1] := 1; end for;
>  B := LLL(B:Delta:=0.999);
>  B := LLL(B:Delta:=0.999, DeepInsertions);
>  EnumerationCost(Lattice(B));
2.77E12
> p:=[1.0: i in [1..65]];
> for i:=10 to 55 do p[i] := (100-i)/90.; end for;
> for i:=56 to 65 do p[i]:=0.5; end for;
> EnumerationCost(Lattice(B):Prune:=p);
1.75E10
> time _:=ShortestVectors(Lattice(B):Prune:=p);
Time: 422.120
```

Assuming that the number of visited tree nodes per time unit remains roughly constant, the execution would have taken around 18 hours without pruning. It is possible to estimate the likeliness of not missing the optimal solution, by looking at the first coefficient returned by `EnumerationCostArray`. Assuming the Gaussian heuristic, this is the ratio between the first coefficients of `EnumerationCostArray` with and without the pruning table.

```
>  t1:= EnumerationCostArray(Lattice(B):Prune:=p)[1];
>  t2:= EnumerationCostArray(Lattice(B))[1];
>  t1/t2;
0.992
```

Experimentally, it seems that pruning coefficients that decrease linearly (except for the first and last indices) provide good trade-offs between efficiency gain and success likeliness.

Example: Parallel Enumeration (ex-d59c02)

There is now an option to use parallel enumeration methods. This does not respect every parameter. It can be turned on with `SetNthreads` and a number of threads (including 1).

```
> SetNthreads(4); // 4 threads
> L := Lattice(LatticeDatabase(),"Leech");
> D := DirectSum([L,L]); // dimension 48
> cpu := Cputime(); t := Time();
> ThetaSeries(D,6); // real time of 1 min
1 + 393120*$.1^4 + 33546240*$.1^6 + O($.1^7)
> Cputime(cpu); Time(t);
210.020 60.740[r]
```

There are situations where parallelization is not beneficial, or even harmful. For instance, when `ShortVectors` returns thousands or millions of vectors, these will need to put serially into a Magma array, and thus any gains from a parallel search will likely be lost. Also, in lattices of small dimension the splitting into parallel parts need not be too efficient.

31.9 Theta Series as Modular Forms

The theta series of an integral lattice L is (the q-expansion of) a modular form whose weight is half the dimension of the lattice, and whose level and nebentypus are determined by the quotient $L^{\#}/L$. The space of forms with given weight, level, and character is finite dimensional, which means the theta series is uniquely characterised as an element of that space from knowledge of finitely many of its coefficients.

The routine described below carries this out explicitly: given an integral lattice, it returns an element of a Magma space of modular forms, Moreover this is done with the least possible effort spent determining coefficients via lattice enumeration. Several ideas are used here, the most important of which is that linear constraints can be obtained from knowing coefficients of the theta series of L *and* of its partial

dual lattices; in practice, one needs a certain number of coefficients in total, and the `EnumerationCost` functionality is useful for balancing how many to compute for each of the duals.

We say L is *q-modular* if it is isomorphic to its qth partial dual L_q. Knowledge that modularities exist is of some use in the algorithm, because q-modularity clearly implies that L and L_q have the same theta series.

Normalisation of theta series: We use a normalisation that is most natural in this context, in order that the theta series is a modular form on $\Gamma_0(N)$ with the level N as small as possible. In this section, the theta series of an even integral lattice L will mean $\Sigma_{v\in L}\, q^{1/2|v|^2}$, while the theta series of an odd integral lattice L will mean $\Sigma_{v\in L}\, q^{|v|^2}$, where $q = e^{2i\pi z}$.

`ThetaSeriesModularFormSpace(L): Lat -> ModFrm`

Given an integral lattice L, this returns the space of modular forms which contains the theta series of L. Note: the theta series is normalised as described above.

`ThetaSeriesModularForm(L): Lat -> ModFrmElt`

```
KnownTheta         : RngSerPowElt                    Default:
KnownDualThetas    : SeqEnum[Tup]                    Default:
KnownModularities  : Set[RngIntElt]                  Default:
ComputeModularities: Set[RngIntElt]                  Default: %%% TO DO:can␣
↪also be BoolElt
```

Given an integral lattice L, this returns the theta series of L as a modular form in the appropriate space: more precisely, in `ThetaSeriesModularFormSpace(L)`. Note: the theta series is normalised as described above.

If some coefficients of the theta series of L are already known, this information may be specified by setting the optional argument `KnownTheta` to be a power series f, indicating that L has theta series equal to f up to the precision of f.

More generally, if coefficients are known for any partial dual lattices, these may be specified by setting `KnownDualThetas`. This argument must be a sequence of tuples of the form $< q, f_q >$, indicating that the qth partial dual L_q has theta series equal to f_q up to the precision of f_q. (*Warning:* care must be taken to use the expected normalisation of the partial duals here.)

If L is known to possess modularities, the optional argument `KnownModularities` may be set equal to any set of integers q such that L is q-modular (as defined above). In addition, `ComputeModularities` may be specified to control whether the function checks for q-modularities (for possible q, that are not listed as `KnownModularities`); its value may be a boolean, or a set of integers.

31.10 Voronoi Cells, Holes, and Covering Radius

The functions in this section compute the Voronoi cell of a lattice around the origin and associated information. Note that the computation of the Voronoi cell is of truly exponential complexity, and therefore the use of these functions is restricted to small dimensions (up to about 10). See [JC98] for the relevant definitions. A lattice to which any of these functions are applied must be an exact lattice (over $\mathbb{Z}$ or $\mathbb{Q}$).

`VoronoiCell(L): Lat -> [ ModTupFldElt ], { SetEnum }, [ ModTupFldElt ]`

Return the Voronoi cell of the lattice L around the origin which is the convex polytope consisting of all the points closer to the origin than to any other lattice point of L. This function returns three values:

(a) A sequence V of vectors which are the vertices of the Voronoi cell.

(b) A set E of pairs, where each pair $\{i, j\}$ represents an edge connecting $V[i]$ and $V[j]$.

(c) A sequence P of vectors defining the relevant hyperplanes. A vector p corresponds to the hyperplane given by $(x, p) = \mathrm{Norm}(p)/2$.

`VoronoiGraph(L): Lat -> GrphUnd`

Return a graph having the vertices and edges of the Voronoi cell of the lattice L as vertices and edges, respectively.

`Holes(L): Lat -> [ ModTupFldElt ]`

Return a sequence of vectors which are the holes of the lattice L. The holes are defined to be the vertices of the Voronoi cell around the origin. Note that this involves computing the Voronoi cell of L around the origin.

`DeepHoles(L): Lat -> [ ModTupFldElt ]`

Return a sequence of vectors which are the deep holes of the lattice L. The deep holes are defined to be the holes of maximum norm and are points of maximum distance to all lattice points. Note that this involves computing the Voronoi cell of L around the origin.

`CoveringRadius(L): Lat -> FldRatElt`

Return the squared covering radius of the lattice L, which is the norm of the deep holes of L. Note that this involves computing the Voronoi cell of L around the origin.

`VoronoiRelevantVectors(L): Lat -> [ ModTupFldElt ]`

Return the Voronoi relevant hyperplanes (as a set of vectors) of the Voronoi cell of L around the origin. Note that this is the same as the third return value of the `VoronoiCell` intrinsic. However, it is usually much faster since it does not compute the Voronoi cell of L. The algorithm employed is [AEVZ02, Section C].

Example: Voronoi Cell of a Perfect Lattice (ex-c4cf07)

We compute the Voronoi cell of a perfect lattice of dimension 6.

```
> L := LatticeWithGram(6, [4, 1,4, 2,2,4, 2,2,1,4, 2,2,1,1,4, 2,2,2,2,2,4]);
> L;
Standard Lattice of rank 6 and degree 6
Inner Product Matrix:
[4 1 2 2 2 2]
[1 4 2 2 2 2]
[2 2 4 1 1 2]
[2 2 1 4 1 2]
[2 2 1 1 4 2]
[2 2 2 2 2 4]
> time V, E, P := VoronoiCell(L);
Time: 1.740
> #Holes(L), #DeepHoles(L), CoveringRadius(L);
782 28 5/2
```

The Voronoi cell has 782 vertices, but only 28 of these are of maximal norm 5/2 and therefore deep holes. We now compute the norms and cardinalities for the shallow holes.

```
> M := MatrixRing(Rationals(), 6) ! InnerProductMatrix(L);
> N := [ (v*M, v) : v in V ];
> norms := Sort(Setseq(Set(N))); norms;
[ 17/9, 2, 37/18, 20/9, 7/3, 5/2 ]
> card := [ #[ x : x in N | x eq n ] : n in norms ]; card;
[ 126, 16, 288, 180, 144, 28 ]
```

So there are 126 holes of norm 17/9, 16 holes of norm 2, etc. We now investigate the Voronoi cell as a polyhedron.

```
> #V, #E, #P;
782 4074 104
> { Norm(L!p) : p in P };
```

(continues on next page)

(continued from previous page)

```
{ 4, 6 }
> #ShortVectors(L, 6);
52
```

The polyhedron which is the convex closure of the holes has 782 vertices, 4074 edges, and 104 faces. The faces are defined by vectors of length up to 6 and all such vectors are relevant (since there are only 104). We finally look at the graph defined by the vertices and edges of the Voronoi cell.

```
> G := VoronoiGraph(L);
> IsConnected(G);
true
> Diameter(G);
8
> Maxdeg(G);
20 ( -1   0 1/2 1/2 1/2   0)
> v := RSpace(Rationals(), 6) ! [ -1, 0, 1/2, 1/2, 1/2, 0 ]; (v*M, v);
5/2
```

The graph is (of course) connected, its diameter is 8 and the vertices of maximal degree 20 are exactly the deep holes.

31.11 Orthogonalization

The functions in this section perform orthogonalization and orthonormalization of lattice bases over the field of fractions of the base ring. Note that this yields a basis orthogonalization of the space in which a lattice embeds; in contrast `OrthogonalDecomposition` returns a decomposition into orthogonal components over the base ring. Basis orthogonalization is equivalent to diagonalization of the inner product matrix of a space.

`Orthogonalize(M): MtrxSpcElt -> MtrxSpcElt, AlgMatElt, RngIntElt`

Given a basis matrix M over a subring R of the real field, compute a matrix N which is row-equivalent over to M over the field of fractions K of R, but whose rows are orthogonal (i.e., NN^{tr} is a diagonal matrix). This function returns three values:

(a) An orthogonalized matrix N in row-equivalent to X over K;

(b) An invertible matrix T in the matrix ring over K whose degree is the number of rows of M such that $TM = N$;

(c) The rank of M.

`Orthogonalization(F): MtrxSpcElt -> MtrxSpcElt, AlgMatElt, RngIntElt`

`OrthogonalizeGram(F): MtrxSpcElt -> MtrxSpcElt, AlgMatElt, RngIntElt`

Given a symmetric $n\times$n matrix F over R, where R is a subring of the real field, compute a *diagonal* matrix G such that $G = TFT^{tr}$ for some invertible matrix T over K, where K is the field of fractions of R. F need not have rank n. This function returns three values:

(a) A diagonal matrix G defined over R;

(b) An invertible $n\times$n matrix T over K such that $G = TFT^{tr}$;

(c) The rank of F.

`OrthogonalizeGram(F, a): MtrxSpcElt, FldAut -> MtrxSpcElt, AlgMatElt, RngIntElt`

Given an $n \times n$ matrix F over a field K of characteristic not 2, which is Hermitian with respect to the automorphism a, compute a *diagonal* matrix G such that $G = TFa(T^{tr})$ for some invertible matrix T over K. F need not have rank n. This function returns three values:

(a) A diagonal matrix G defined over K;

(b) An invertible $n\times$n matrix T over K such that $G = TFa(T^{tr})$;

(c) The rank of F.

`Orthogonalize(L): Lat -> Lat, AlgMatElt`

For a lattice L, return a new lattice having the same Gram matrix as L but embedded in an ambient space with diagonal inner product matrix.

`Orthonormalize(M, K): MtrxSpcElt, Fld -> AlgMatElt`

`Cholesky(M, K): MtrxSpcElt, Fld -> AlgMatElt`

`Orthonormalize(M): MtrxSpcElt -> AlgMatElt`

`Cholesky(M): MtrxSpcElt -> AlgMatElt`

For a symmetric, positive definite matrix M, and a real field K, return a lower triangular matrix T over K such that $M = TT^{tr}$. The algorithm must take square roots so the result is returned as a matrix over the real field K. If the real field K is omitted, K is taken to be the default real field. Note that this function takes a Gram matrix M, *not* a basis matrix as in the previous functions.

`Orthonormalize(L, K): Lat, FldRe -> AlgMatElt`

`Cholesky(L, K): Lat, FldRe -> AlgMatElt`

`Orthonormalize(L): Lat -> AlgMatElt`

`Cholesky(L): Lat -> AlgMatElt`

Given a lattice L with Gram matrix F, together with a real field K, return a new lattice over K which has the same Gram matrix F as L but has the standard Euclidean inner product. (This will involve taking square roots so that is why the result must be over a real field.) The argument for the real field K may be omitted, in which case K is taken to be the current default real field. This function is equivalent to the invocation `LatticeWithBasis(Orthonormalize(GramMatrix(L), K))`. It is sometimes more convenient to work with the resulting lattice since it has the standard Euclidean inner product.

Example: Orthogonalizing the Dual Coxeter-Todd Lattice `(ex-452db1)`

As an example for a lattice with non-trivial basis and inner product matrices we choose the dual lattice of the 12-dimensional Coxeter-Todd lattice. We compute the inner products of all pairs of shortest vectors and notice that this gets faster after changing to an isomorphic lattice with weighted standard Euclidean inner product.

```
> L := Dual(CoordinateLattice(Lattice("Kappa", 12)));
> SL := ShortestVectors(L);
> SL := SL cat [ -v : v in SL ]; #SL;
756
> time { (v,w) : v,w in SL };
{ -4, -2, -1, 0, 1, 2, 4 }
Time: 7.120
> M := Orthogonalize(L);
> SM := ShortestVectors(M);
> SM := SM cat [ -v : v in SM ]; #SM;
756
> time { (v,w) : v,w in SM };
{ -4, -2, -1, 0, 1, 2, 4 }
Time: 1.300
```

31.12 Testing Matrices for Definiteness

The functions in this section test matrices for positive definiteness, etc. They may applied to any symmetric matrix over a real subring (i.e., $\mathbb{Z}$, $\mathbb{Q}$, or a real field), though the `SemiDefinite` functions must be over $\mathbb{Z}$ or $\mathbb{Q}$. Each function works by calling the function `OrthogonalizeGram` on its argument

and then determining whether the resulting diagonal matrix has the appropriate form. Over a real field, a numerical check is made that the matrix has enough stability to determine definiteness.

IsPositiveDefinite(F): Mtrx -> BoolElt

Given a symmetric matrix F over the rationals or integers or a real field, return whether F is positive definite, i.e., whether $vFv^{tr} > 0$ for all non-zero vectors $v \in \mathbb{R}^n$. Over a real field, it will fail in numerically unstable situations.

IsPositiveSemiDefinite(F): Mtrx -> BoolElt

Given a symmetric matrix F over the rationals or integers, return whether F is positive semi-definite, i.e., whether $vFv^{tr} \geq 0$ for all non-zero vectors $v \in \mathbb{R}^n$.

IsNegativeDefinite(F): Mtrx -> BoolElt

Given a symmetric matrix F over the rationals or integers or a real field, return whether F is negative definite, i.e., whether $vFv^{tr} < 0$ for all non-zero vectors $v \in \mathbb{R}^n$. Over a real field, it will fail in numerically unstable situations.

IsNegativeSemiDefinite(F): Mtrx -> BoolElt

Given a symmetric matrix F over the rationals or integers, return whether F is negative semi-definite, i.e., whether $vFv^{tr} \leq 0$ for all non-zero vectors $v \in \mathbb{R}^n$.

Signature(F): Mtrx -> RngIntElt, RngIntElt, RngIntElt

Given a symmetric matrix F over the rationals or integers, return the signature of F, i.e. the number of positive, negative, and zero eigenvalues.

NumericalSignature(M): Mtrx -> RngIntElt, RngIntElt

Given a symmetric matrix over a real field, return its signature, that is, the number of positive and negative eigenvalues. Fails if (at least) one of the eigenvalues is too close to zero.

31.13 Genera and Spinor Genera

The genus of an exact lattice has a distinct type `SymGen` which holds the local data defining the genus. Each genus consists of 2^n spinor genera, for some integer n, typically 1. The spinor genera share the same type `SymGen`. Unlike the genus, the spinor genus is not determined solely by the local data of the genus, so a cached representative is necessary to define the spinor class.

Equality testing of genera is fast, since this requires only a comparison of the canonical local information. It is also possible to enumerate representatives of all equivalences classes in a genus or spinor genus. This is done by a process of exploration of the p-neighbour graph, for an appropriate prime p. The neighbouring functions can be applied to individual lattices to find p-neighbours or the closure under the p-neighbour process. Functions for computing and comparing the local p-adic equivalence classes of lattices, mediated by the type `SymGenLoc`.

31.13.1 Genus Constructions

Genus(L): Lat -> SymGen

Genus(G): SymGen -> SymGen

Given an exact lattice L or a spinor genus G this function returns the genus of L. If given a genus the function returns G itself.

Genera(r, s, d): RngIntElt, RngIntElt, RngIntElt -> [SymGen]

OnlyEven: BoolElt	Default: true

Given a signature (r, s) and a determinant d, returns all valid genus symbols representing lattices with this signature and this determinant. If `OnlyEven` is `true`, return only the genera representing even lattices.

SpinorGenus(L): Lat -> SymGen

Given an exact lattice L, returns the spinor genus of L.

`SpinorGenera(G): SymGen -> [ SymGen ]`

Given a genus G, returns the sequence of spinor genera. If G is a spinor genus, then this function returns the sequence consisting of G itself.

`Genus(r, s, d, S): RngIntElt, RngIntElt, RngIntElt, [ SymGenLoc ] -> SymGen`

Given a signature (r, s), a determinant d, and a sequence of local Genus symbols $S = \{G_p\}_p$, return the corresponding global genus G.

31.13.2 Local Genus Constructions

`LocalGenus(L, p): Lat, RngIntElt -> SymGenLoc`

Returns the local genus symbol of L at p.

`LocalGenus(A, p): AlgMatElt[RngInt], RngIntElt -> SymGenLoc`

Returns the local genus symbol of a lattice with Gram matrix A at p.

`CanonicalLocalGenus(G): SymGenLoc -> SymGenLoc`

Returns the canonical p-adic genus symbol of G. Only relevant at $p = 2$.

`LocalGenus(p, v, r, d): RngIntElt, [ RngIntElt ], [ RngIntElt ], [ RngElt ] -> SymGenLoc`

```
Parities: [ RngElt ]                    Default: []
Oddities: [ RngElt ]                    Default: []
```

Given a prime p, and sequence of valuations $v_1, \ldots, v_n$, ranks $r_1, \ldots, r_n$, and determinants $d_1, \ldots, d_n$ (which should be coercible to $\mathbb{Z}/p\mathbb{Z}$), construct the p-adic genus symbol having Jordan components with these ranks, valuations, and determinants. A lattice L in the constructed local genus will have a p-adic Jordan decomposition of the form $L_p = \bigoplus_i p^{v_i} L_i$, where L_i is of rank r_i and determinant d_i. If $p = 2$, the optional parameters `Parities` and `Oddities` should be specified, in order to specify the parity and oddity of each component.

`LocalGenera(n, d, p): RngIntElt, RngIntElt, RngIntElt -> [ SymGenLoc ]`

Return all p-adic genus symbols of rank n and determinant d, for an odd prime p.

`pAdicDiagonalization(L, p): Lat, RngIntElt -> Lat`

A lattice diagonalized p-adically equivalent to L; if $p = 2$ then the diagonalized form may have Jordan blocks of dimension 2.

`JordanDecomposition(L, p): Lat, RngIntElt -> List, List, SeqEnum`
`JordanDecomposition(L, P): Lat, RngInt -> List, List, SeqEnum`

Given a lattice L and a prime ideal $P = p\mathbb{Z}$ of $\mathbb{Z}$, a Jordan decomposition of L_p is an orthogonal decomposition of L_p into modular sublattices $L_1, \ldots, L_r$ such that the p-adic valuations of the scales of these sublattices are strictly increasing.

The first return value is a list of matrices $B_1, \ldots, B_r$ over $\mathbb{Q}$. The row spans of these lattices over $\mathbb{Z}_p$ yield a Jordan decomposition of L_p. The second and third return value list the Gram matrices and the p-adic valuations of the scales of the Jordan blocks respectively.

`pAdicJordanDecomposition(L, p): Lat, RngIntElt -> AlgMatElt, AlgMatElt`

Given a lattice L and a prime p, return a canonical p-adic Jordan decomposition of its Gram matrix, along with the change of basis matrix. Returns matrices J, B over $\mathbb{Z}_p$ such that $BMB^{tr} = J$, where M is the Gram matrix of L.

Every Jordan block of J is of the following form. If p is odd, it is diagonal of the form $\langle 1, 1, ..., \ldots, 1, u\rangle$ where u is a predetermined non-square class. If $p = 2$, it is of the form $H^m \oplus A^\epsilon \oplus \langle 1, 1, \ldots, 1, u\rangle$, where H is a hyperbolic plane, A is the anisotropic form $x^2 + xy + y^2$, $m \geq 0$ and $\epsilon \in \{0, 1\}$.

`LocalModification(L, G, p): Lat, AlgMatElt[RngInt], RngIntElt -> Lat`

An integral lattice L' in the ambient space of L which is locally isometric to L at every prime other than p, and at p, locally isometric to a lattice with the Gram matrix G.

31.13.3 Invariants of Genera and Spinor Genera

`Primes(G): SymGen -> [ RngIntElt ]`

Returns the primes that appear in G. These are the primes for which G_p is nontrivial.

`Dimension(G): SymGen -> RngIntElt`

`Rank(G): SymGen -> RngIntElt`

The rank of lattice in the genus G.

`Signature(G): SymGen -> RngIntElt, RngIntElt`

The signature of lattices in G. Returns a pair of integers (r, s) such that the matrix representing the bilinear form obtaind the eigenvalue 1 with multiplicity r, and the eigenvalue -1 with multiplicity s.

`Representative(G): SymGen -> Lat`

Returns a representative lattice for the genus symbol G.

`RationalRepresentative(G): SymGen -> AlgMatElt[FldRat]`

Returns a Gram matrix representing the ambient quadratic vector space (over $\mathbb{Q}$) of a lattice representing G.

`IsEven(G): SymGen -> BoolElt`

Returns `true` if G represents even lattices.

`IsSpinorGenus(G): SymGen -> BoolElt`

Returns `true` if and only if G is a spinor genus. This is the negation of `IsGenus(G)`.

`IsGenus(G): SymGen -> BoolElt`

Returns `true` if and only if G is a genus. This is the negation of `IsSpinorGenus(G)`.

`Determinant(G): SymGen -> Lat`

Returns the determinant of the genus symbol G.

`LocalGenera(G): SymGen -> Lat`

Returns the sequence of p-adic genera of the genus symbol G.

`ConwaySymbol(G): SymGen -> MonStgElt`

Returns the Conway symbol of G.

`G1 eq G2: SymGen, SymGen -> BoolElt`

Given two genus symbols, return `true` if and only if they represent the same genus. This computation is fast for genera, but currently for spinor genera invokes a call to `Representatives`.

`# G: SymGen -> RngIntElt`

The number of isometry classes in the genus or spinor genus G.

Enumeration of isometry classes is done by an explicit call to `Representatives`, so that `#G` is an expensive computation.

`SpinorCharacters(G): SymGen -> [ GrpDrchElt ]`

Return the spinor characters of the genus symbol G as a sequence of Dirichlet characters whose kernels intersect exactly in the group of automorphous numbers. Consult Conway and Sloane [JC98] for precise definitions and significance of the spinor kernel and automorphous numbers.

`SpinorGenerators(G): SymGen -> [ RngIntElt ]`

Return the spinor generators of the genus symbol G as a sequence of primes which generate the group of spinor norms. The primes generate a group dual to that generated by the spinor characters.

`AutomorphousClasses(L, p): Lat, RngIntElt -> { RngIntElt }`

`AutomorphousClasses(G, p): SymGen, RngIntElt -> { RngIntElt }`

A set of integer representatives of the p-adic square classes in the image of the spinor norm of the lattice L (respectively the genus symbol G).

`IsSpinorNorm(G, p): SymGen, RngIntElt -> { RngIntElt }`

Returns `true` if and only if p is coprime to 2 and the determinant, and p is the norm of an element of the spinor kernel of G.

31.13.4 Invariants of p-adic Genera

`Prime(G): SymGenLoc -> RngIntElt`

Return the prime p for which G represents the p-adic genus.

`Representative(G): SymGenLoc -> Lat`

Returns a canonical representative lattice of the p-adic genus G, with Gram matrix in Jordan form. For odd p the Jordan form is diagonalized.

`Determinant(G): SymGenLoc -> RngIntElt`

This function returns a canonical p-adic representative of the determinant of the p-adic genus G. The determinant is well-defined only up to squares.

`Dimension(G): SymGenLoc -> RngIntElt`

`Rank(G): SymGenLoc -> RngIntElt`

Return the dimension of the p-adic genus G.

`Oddity(G): SymGenLoc -> RngIntResElt`

Returns the oddity (in $\mathbb{Z}/8\mathbb{Z}$) of the 2-adic genus symbol G.

`Excess(G): SymGenLoc -> RngIntResElt`

Returns the p-excess (in $\mathbb{Z}/8\mathbb{Z}$) of the p-adic genus symbol G.

`Compartments(G): SymGenLoc -> [ [ RngIntElt ] ]`

Returns the compartments of the 2-adic genus symbol G. Compartments are maximal intervals where all factors are of scaled type I.

`Trains(G): SymGenLoc -> [ [ RngIntElt ] ]`

Returns the trains of the 2-adic genus symbol G. Trains are maximal intervals having the property that for each pair of adjacent forms at least one is of scaled type I.

`ConwaySymbol(G): SymGenLoc -> MonStgElt`

`Global: BoolElt`	`Default: false`

Returns the Conway symbol of G. If `Global` is `true`, returns its representation as part of the Conway symbol of a global genus.

`IsEven(G): SymGenLoc -> BoolElt`

Returns `true` if G represents even lattices. If p is odd, every lattice is even.

`GramMatrix(G): SymGenLoc -> AlgMatElt[RngInt]`

Returns a Gram matrix for a lattice L such that L_p is in G_p.

`G1 eq G2: SymGenLoc, SymGenLoc -> BoolElt`

Given local genus symbols $G1$ and $G2$, return `true` if and only if they have the same prime and the same canonical Jordan form.

`# G: SymGenLoc -> RngIntElt`

The number of Jordan components in the p-adic genus symbol G.

31.13.5 Neighbour Relations and Graphs

`Neighbour(L, v, p): Lat, LatElt, RngIntElt -> Lat`

`Neighbor(L, v, p): Lat, LatElt, RngIntElt -> Lat`

Let L be an integral lattice, p a prime which does not divide Determinant(L) and v a vector in $L \setminus pL$ with $(v, v) \in p^2\mathbb{Z}$. The p-neighbour of L with respect to v is the lattice generated by L_v and $p^{-1}v$, where $L_v := \{x \in L | (x, v) \in p\mathbb{Z}\}$.

See [Kne57] for the original definition and [SP91] for a generalization of the neighbouring method.

`Neighbours(L, p): Lat, RngIntElt -> Lat`

`Neighbors(L, p): Lat, RngIntElt -> Lat`

For an integral lattice L and prime p, returns the sequence of p-neighbours of L.

`NeighbourClosure(L, p): Lat, RngIntElt -> Lat`

`NeighborClosure(L, p): Lat, RngIntElt -> Lat`

```
NaturalAction: BoolElt                  Default: false
Proper       : BoolElt                  Default: false
UseAuto      : BoolElt                  Default: true
ThetaPrec    : RngIntElt                Default: 12
UseMass      : BoolElt                  Default: true
Bound        : RngIntElt                Default: 2^32
Depth        : RngIntElt                Default: -1
```

For an integral lattice L and prime p, returns the sequence of lattices obtained by transitive closure of the p-neighbours of L. If `NaturalAction` is set to `true`, the representatives returned lie in the same quadratic space as L. If `Proper` is set to `true`, returns representatives for proper isometry classes. If `UseMass` is set to `false`, does not use the mass formula as a stopping condition, and compute the entire Neighbor closure. `ThetaPrec` determines the precision to which theta series of the representatives will be computed.

Note that neighbours with respect to two vectors v_1, v_2 whose images in L/pL lie in the same projective orbit of Aut(L) on L/pL are isometric. Therefore, as long as `UseAuto` is set to `true`, only projective orbit representatives of the action of Aut(L) on L/pL are used. The large number of orbits restricts the complexity of this algorithm, hence the function gives an error if $p^{\mathrm{Rank}(L)}$ is greater than `Bound`, by default set to 2^{32}. The `Depth` parameter is used to control the depth in the calls to `AutomorphismGroup` and `IsIsometric`.

`GenusRepresentatives(L): Lat -> [ Lat ], Assoc`

`SpinorRepresentatives(L): Lat -> [ Lat ], Assoc`

`Representatives(G): SymGen -> [ Lat ], Assoc`

`Representatives(G): SymGen -> [ LatNF ], Assoc`

```
NaturalAction: BoolElt                  Default: false
Proper       : BoolElt                  Default: false
ThetaPrec    : RngIntElt                Default: 12
Bound        : RngIntElt                Default: 2^32
Depth        : RngIntElt                Default: -1
```

For an exact lattice L with genus or spinor genus G, this function enumerates the isometry classes in G by constructing the p-neighbour closure (up to isometry). This construction used using an appropriate prime or primes p not dividing the determinant of L. For the genus, sufficiently many primes p are chosen to generate the full image, modulo the spinor kernel, of each character defining the spinor kernel. The parameters are exactly as for the `NeighbourClosure` function.

`PositiveDefiniteQuadraticLattices(n, d): RngIntElt, RngIntElt -> [ [ Lat ] ]`

`TernaryQuadraticLattices(d): RngIntElt -> [ [ Lat ] ]`

`QuaternaryQuadraticLattices(d): RngIntElt -> [ [ Lat ] ]`

`QuinaryQuadraticLattices(d): RngIntElt -> [ [ Lat ] ]`

Given a positive integer d, returns representatives for all isometry classes of positive definite lattices with discriminant (resp. half-discriminant for odd rank) d, grouped by genus.

`AdjacencyMatrix(G, p): SymGen, RngIntElt -> AlgMatElt`

For a genus or spinor genus G, this function determines the adjacency matrix of the p-neighbour graph on the representative classes for G. The integer p must be prime, and if G is a spinor genus, then an error ensues if p is not an automorphous number for G.

Example: Constructing E8 as a 2-Neighbour (ex-4d42d6)

We construct the root lattice E_8 (the unique even unimodular lattice of dimension 8) as a 2-neighbour of the 8-dimensional standard lattice.

```
> Z8 := StandardLattice(8);
> v := Z8 ! [1,1,1,1,1,1,1,1];
> E8 := Neighbour(Z8, v, 2);
> E8;
Lattice of rank 8 and degree 8
Basis:
( 2  0  0  0  0  0  0  2)
( 2  0  0  0  0  0  0 -2)
( 1  1 -1  1  1 -1  1  1)
( 1  1 -1 -1 -1 -1 -1 -1)
( 1 -1 -1 -1 -1 -1  1 -1)
( 0  0  2  0  0  0  0  2)
( 0  0  0  0  2  0  0  2)
( 0  0  0  0  0  2  0  2)
Basis denominator: 2
```

The so-obtained lattice is in fact identical to the one returned by the standard construction.

```
> L := Lattice("E", 8);
> L;
Lattice of rank 8 and degree 8
Basis:
( 4  0  0  0  0  0  0  0)
(-2  2  0  0  0  0  0  0)
( 0 -2  2  0  0  0  0  0)
( 0  0 -2  2  0  0  0  0)
( 0  0  0 -2  2  0  0  0)
( 0  0  0  0 -2  2  0  0)
( 0  0  0  0  0 -2  2  0)
( 1  1  1  1  1  1  1  1)
Basis Denominator: 2
> E8 eq L;
true
```

Example: Genus of the Coxeter-Todd Lattice (ex-20b1ce)

In this example we enumerate representatives for the genus of the Coxeter-Todd lattice, performing the major steps manually. The whole computation can be done simply by calling the `GenusRepresentatives` function but the example illustrates how the function actually works.

We use a combination of the automorphism group, isometry, and neighbouring functions. The idea is that the neighbouring graph spans the full genus which therefore can be computed by successively generating neighbours and checking them for isometry with already known ones. The automorphism group comes into play, since neighbours with respect to vectors in the same projective orbit under the automorphism group are isometric.

```
> L := CoordinateLattice(Lattice("Kappa", 12));
> G := AutomorphismGroup(L);
> G2 := ChangeRing(G, GF(2));
```

(continues on next page)

(continued from previous page)

```
> O := LineOrbits(G2);
> [ Norm(L!Rep(o).1) : o in O ];
[ 4, 8, 10 ]
```

Hence only the first and second orbits give rise to a 2-neighbour. To obtain an even neighbour, the second vector has to be adjusted by an element of $2 * L$ such that it has norm divisible by 8.

```
> v1 := L ! Rep(O[1]).1;
> v1 +:= 2 * Rep({ u : u in Basis(L) | (v1,u) mod 2 eq 1 });
> v2 := L ! Rep(O[2]).1;
> Norm(v1), Norm(v2);
16 8
> L1 := Neighbour(L, v1, 2);
> L2 := Neighbour(L, v2, 2);
> bool := IsIsometric(L, L1); bool;
true
> bool := IsIsometric(L, L2); bool;
false
```

So we obtain only one non-isometric even neighbour of L. To obtain the full genus we can now proceed with $L2$ in the same way, and do this with the following function `EvenGenus`. Note that this function is simply one component of the function `GenusRepresentatives`.

```
> function EvenGenus(L)
>       // Start with the lattice L
>       Lambda := [ CoordinateLattice(LLL(L)) ];
>       cand := 1;
>       while cand le #Lambda do
>           L := Lambda[cand];
>           G := ChangeRing( AutomorphismGroup(L), GF(2) );
>           // Get the projective orbits on L/2L
>           O := LineOrbits(G);
>           for o in O do
>               v := L ! Rep(o).1;
>               if Norm(v) mod 4 eq 0 then
>                   // Adjust the vector such that its norm is divisible by 8
>                   if not Norm(v) mod 8 eq 0 then
>                       v +:= 2 * Rep({ u : u in Basis(L) | (v,u) mod 2 eq 1 });
>                   end if;
>                   N := LLL(Neighbour(L, v, 2));
>                   new := true;
>                   for i in [1..#Lambda] do
>                       if IsIsometric(Lambda[i], N) then
>                           new := false;
>                           break i;
>                       end if;
>                   end for;
>                   if new then
>                       Append(~Lambda, CoordinateLattice(N));
>                   end if;
>               end if;
>           end for;
>           cand +:= 1;
>       end while;
>       return Lambda;
> end function;
```

(continues on next page)

(continued from previous page)

```
>
> time Lambda := EvenGenus(L);
Time: 9.300
> #Lambda;
10
> [ Minimum(L) : L in Lambda ];
[ 4, 2, 2, 2, 2, 2, 2, 2, 2, 2 ]
> &+[ 1/#AutomorphismGroup(L) : L in Lambda ];
4649359/4213820620800
```

We see that the genus consists of 10 classes of lattices where only the Coxeter-Todd lattice has minimum 4 and get the mass of the genus as 4649359/4213820620800.

31.14 Attributes of Lattices

This section lists various attributes of lattices which can be examined and set by the user. This allows low-level control of information stored in lattices. Note that when an attribute is set, only minimal testing can be done on the value so if an incorrect value is set, unpredictable results may occur. Note also that if an attribute is not set, referring to it in an expression (using the ` operator) will *not* trigger the calculation of it (while intrinsic functions do); rather an error will ensue. Use the `assigned` operator to test whether an attribute is set.

`L`Minimum: Lat -> RngElt`

The attribute for the minimum of a rational or integer lattice L. If the attribute `L`Minimum` is examined, either the minimum is known so it is returned or an error results. If the attribute `L`Minimum` is set by assignment, it must be a positive number equal to the minimum of the lattice. Magma will not check that this is correct since that may be very time-consuming. If the attribute is already set, the new value must be the same as the old value.

`L`MinimumBound: Lat -> RngElt`

The attribute for an upper bound to the minimum of a rational or integer lattice L. If the attribute `L`MinimumBound` is examined, either an upper bound to the minimum is known so it is returned or an error results. If the attribute `L`MinimumBound` is set by assignment, it must be a positive number greater than or equal to the minimum of the lattice. Magma will not check that this is correct since that may be very time-consuming. If the attribute is already set, the new value must be the same as or smaller than the old value.

31.15 Database of Lattices

Magma includes a database containing most of the lattices explicitly presented in the Catalogue of Lattices maintained by Neil J.A. Sloane and Gabriele Nebe [NS01b].

Many standard lattices included in the Sloane & Nebe catalogue are not in the database as they may be obtained by applying Magma's standard lattice creation functions. Also omitted from the database are a small number of catalogued lattices defined over rings other than $\mathbb{Z}$ or $\mathbb{Q}$.

The information available for any given lattice in the catalogue varies considerably. A similar variety is found in the Magma database version, although some data (generally either easily computable or rarely available in the catalogue) is omitted.

Where the Magma database does retain data, it is not altered from the data in the catalogue. Thus the caveat which comes with that catalogue remains relevant: "Warning! Not all the entries have been checked!"

A second version of the Lattice Database has been made available with Version 2.16 of Magma. It adds a few more lattices, contains more information about automorphism groups, and adds Θ-series as attributes. Furthermore, it removes some duplicates. The user should be warned that the numbering of

lattices (and naming in some cases) differs between the two versions. The newer version, however, does not contain information about any Hermitian structure at the current time.

The entries of the database can be accessed in three ways:

(i) the i-th entry of the database can be requested;

(ii) the i-th entry of a particular dimension d can be specified;

(iii) the desired entry can be denoted by its name N. This name is specified exactly as in the catalogue, including all punctuation and whitespace. In the rare event that two or more entries share a single name, particular entries may be distinguished by supplying an integer i in addition to N, to denote the i-th entry with name N.

31.15.1 Creating the Database

`LatticeDatabase() -> DB`

This function returns a database object which contains information about the database.

31.15.2 Database Information

This section gives the functions that enable the user to find out what is in the database.

`# D: DB -> RngIntElt`

`NumberOfLattices(D): DB -> RngIntElt`

Returns the number of lattices stored in the database.

`LargestDimension(D): DB -> RngIntElt`

Returns the largest dimension of any lattice in the database.

`NumberOfLattices(D, d): DB, RngIntElt -> RngIntElt`

Returns the number of lattices of dimension d stored in the database.

`NumberOfLattices(D, N): DB, MonStgElt -> RngIntElt`

Returns the number of lattices named N stored in the database. (This should always be 1 now).

`LatticeName(D, i): DB, RngIntElt -> MonStgElt, RngIntElt`

Return the name and dimension of the i-th entry of the database D.

`LatticeName(D, d, i): DB, RngIntElt, RngIntElt -> RecMonStgElt, RngIntElt`

Return the name and dimension of the i-th entry of dimension d of the database D.

`LatticeName(D, N): DB, MonStgElt -> RecMonStgElt, RngIntElt`

Return the name and dimension of the first entry of the database with name N.

`LatticeName(D, N, i): DB, MonStgElt, RngIntElt -> RecMonStgElt, RngIntElt`

Return the name and dimension of the i-th entry of the database with name N.

Example: Names in the Lattice Database (ex-8a7900)

We find out the names of the database entries.

```
> D := LatticeDatabase();
> NumberOfLattices(D);
699
```

The database contains 699 lattices. We get the set of all names in the database.

```
> names := {LatticeName(D,i): i in [1..#D]};
> #names; // No duplicate names anymore
699
> Random(names);
```

(continues on next page)

(continued from previous page)

```
S4(5):2
> NumberOfLattices(D, "S4(5):2");
1
```

31.15.3 Accessing the Database

The following functions retrieve lattice information from the database.

`Lattice(D, i: parameters): DB, RngIntElt -> Lattice`
`Lattice(D, d, i: parameters): DB, RngIntElt, RngIntElt -> Lattice`
`Lattice(D, N: parameters): DB, MonStgElt -> Lattice`
`Lattice(D, N, i: parameters): DB, MonStgElt, RngIntElt -> Lattice`

```
TrustAutomorphismGroup: Bool                    Default: true
```

Returns the i-th entry (of dimension d or name N) from the database D as a lattice L.

If the `TrustAutomorphismGroup` parameter is assigned `false`, then any data which claims to be the automorphism group will not be stored in L.

`LatticeData(D, i): DB, RngIntElt -> Rec`
`LatticeData(D, d, i): DB, RngIntElt, RngIntElt -> Rec`
`LatticeData(D, N): DB, MonStgElt -> Rec`
`LatticeData(D, N, i): DB, MonStgElt, RngIntElt -> Rec`

Returns a record which contains all the information about the i-th lattice stored in the database D (of dimension d or name N). The automorphism group is returned separately from the lattice and not stored in it.

Example: Looking Up a Lattice in the Database (ex-7db624)

We look up a lattice in the database. There are 19 lattices of dimension 6 in the database. We get the 10th.

```
> D := LatticeDatabase();
> NumberOfLattices(D, 6);
19
> L := Lattice(D, 6, 10);
> L;
Standard Lattice of rank 6 and degree 6
Minimum: 4
Inner Product Matrix:
[4 1 2 2 2 2]
[1 4 2 2 2 2]
[2 2 4 1 2 2]
[2 2 1 4 2 2]
[2 2 2 2 4 1]
[2 2 2 2 1 4]
```

There may be more information stored than just what is returned by the `Lattice` function. We get the record containing all the stored lattice data.

```
> R := LatticeData(D, 6, 10);
> Format(R);
recformat<name, dim, lattice, minimum, kissing_number,
is_integral, is_even, is_unimodular, is_unimodular_hermitian,
modularity, group_names, group, group_order,
hermitian_group_names, hermitian_group, hermitian_group_order,
hermitian_structure>
```

This lists all possible fields in the record. They may or may not be assigned for any particular lattice.

```
> R`lattice eq L;
true
> R`name;
A6,1
> assigned R`kissing_number;
true
> R`kissing_number;
42
> assigned R`group;
false
> A := AutomorphismGroup(L);
> A : Minimal;
MatrixGroup(6, Integer Ring) of order 96 = 2^5 * 3
```

The result of the `Lattice` call is equal to the `lattice` field of the data record. The kissing number was stored, but the automorphism group wasn't. We computed the group (as a matrix group over the integers) and found it has order 96.

31.15.4 Hermitian Lattices

There are a few facilities for computing with Hermitian lattices over an imaginary quadratic field or a quaternion algebra. However, these functions apply to a Gram matrix, and not a lattice *per se*. The main application is for automorphism groups that preserve a structure.

`HermitianTranspose(M): Mtrx -> Mtrx`

`QuaternionicTranspose(M): Mtrx -> Mtrx`

Given a matrix over an imaginary quadratic field or a quaternion algebra, return the conjugate transpose.

`ExpandBasis(M): Mtrx -> Mtrx`

`QuaternionicBasis(M): Mtrx -> Mtrx`

Given a matrix over an imaginary quadratic field or a quaternion algebra, expand it to a basis over the rationals.

`HermitianAutomorphismGroup(M): Mtrx -> GrpMat`

`QuaternionicAutomorphismGroup(M): Mtrx -> GrpMat`

Given a conjugate symmetric Gram matrix, compute the automorphism group.

Various functions for matrix groups over associative algebras are available here, such as `CharacterTable` and `IsConjugate` which simply use a re-writing over the rationals, and `InvariantForms` which after using `GHom` needs to restrict to elements fixed by the quaterionic structure. Finally, there is `QuaternionicGModule` which will split a G-module over a quaternionic structure.

`InvariantForms(G): GrpMat -> SeqEnum`

Given a matrix group over an associative algebra or an imaginary quadratic field, return a basis for the forms fixed by it.

`QuaternionicGModule(M, I, J): ModGrp, AlgMatElt, AlgMatElt -> ModGrp`

Given a G-module M and I and J in the endomorphism algebra that anti-commute and whose squares are scalars, write G over the quaternionic structure given by I and J.

`MooreDeterminant(M): Mtrx -> Mtrx`

Given a conjugate-symmetric matrix over a quaternion algebra, compute the Moore determinant. This is the "normal" determinant, which is well-defined here because all the diagonal elements are rational, and thus there is no ambiguity between left/right division.

Example: Hermitian Coxeter-Todd Lattice (ex-a41728)

We construct the Coxeter-Todd lattice over $Q_{3,\infty}$ starting with the group $SU(3,3)$.

```
> G := SU(3, 3);
> chi := CharacterTable(G)[2];
> M := GModule(chi,Integers());
> E := EndomorphismAlgebra (M);
> while true do
>     r := &+[Random([-2..2])*E.i : i in [1..4]];
>     if r^2 eq -1 then break; end if;
> end while;
> while true do
>     s := &+[Random([-2..2])*E.i : i in [1..4]];
>     if s^2 eq -3 and r*s eq -s*r then break; end if;
> end while;
> MM := QuaternionicGModule(M, r, s);
> Discriminant(BaseRing(MM));
3
> MG := MatrixGroup(MM);
> IF := InvariantForms(MG); IF;
[
    [1 -1/2*i + 1/6*k 1/3*k]
    [1/2*i - 1/6*k 1 -1/3*j]
    [-1/3*k 1/3*j 1]
]
> assert IsIsomorphic(G, MG);
```

Example: Quaternionic Automorphism Group of the Leech Lattice (ex-723012)

We compute the quaternionic automorphism group for the Leech lattice.

```
> A<i,j,k> := QuaternionAlgebra<Rationals()|-1,-1>;
> v := [];
> v[1] := [2+2*i,0,0,0,0,0]; /* from Wilson's paper */
> v[2] := [2,2,0,0,0,0];
> v[3] := [0,2,2,0,0,0];
> v[4] := [i+j+k,1,1,1,1,1];
> v[5] := [0,0,1+k,1+j,1+j,1+k];
> v[6] := [0,1+j,1+j,1+k,0,1+k];
> V := [Vector(x) : x in v];
> W := [Vector([Conjugate(x) : x in Eltseq(v)]): v in V];
> M6 := Matrix(6,6,[(V[i],W[j])/2 : i,j in [1..6]]); /* 6-dim over A */
> time Q := QuaternionicAutomorphismGroup(M6);
> assert #Q eq 503193600;
```

The same can be done for the Coxeter-Todd lattice.

```
> A<i,j,k> := QuaternionAlgebra<Rationals()|-1,-3>;
> a := (1+i+j+k)/2;
> M3 := Matrix(3,3,[2,a,-1, Conjugate(a),2,a, -1,Conjugate(a),2]);
> time Q := QuaternionicAutomorphismGroup(M3);
> assert #Q eq 12096;
```

One can also compute the automorphism group over the Eisenstein field, using `InvariantForms` on the realisation of this group as `ShephardTodd(34)`.

```
> G := ShephardTodd(34);
> IF := InvariantForms(G); // scaled Coxeter-Todd over Q(sqrt(-3))
```

(continues on next page)

(continued from previous page)

```
> A := HermitianAutomorphismGroup(IF[1]);
> assert IsIsomorphic(A,G);
```

31.16 Bibliography

- [AEVZ02] E. Agrell, T. Eriksson, A. Vardy, and K. Zeger. Closest point search in lattices. *IEEE Transactions on Information Theory*, 48(8):2201–2214, 2002.
- [Ajt98] Miklós Ajtai. The Shortest Vector Problem in L2 is NP-hard for Randomized Reductions (Extended Abstract). In *Proceedings of the 30th Symposium on the Theory of Computing (STOC 1998)*, 10–19. ACM, 1998.
- [Akh02] Ali Akhavi. Random lattices, threshold phenomena and efficient reduction algorithms. *Theoretical Computer Science*, 287(2):359–385, 2002.
- [dW87] Benne M.M. de Weger. Solving exponential Diophantine equations using lattice basis reduction algorithms. *J. Number Th.*, 26:325–367, 1987.
- [FP83] U. Fincke and M. Pohst. A procedure for determining algebraic integers of given norm. In *EUROCAL*, volume 162 of LNCS, 194–202. Springer, 1983.
- [HS07] Guillaume Hanrot and Damien Stehlé. Improved Analysis of Kannan's Shortest Lattice Vector Algorithm (Extended Abstract). In *Advances in cryptology—CRYPTO 2007*, volume 4622 of LNCS, 170–186. Springer, 2007.
- [JC98] N.J.A. Sloane J.H. Conway. *Sphere Packings, Lattices and Groups*. Volume 290 of Grundlehren der Mathematischen Wissenschaften. Springer, New York–Berlin–Heidelberg, 3rd edition, 1998.
- [Kan83] R. Kannan. Improved algorithms for integer programming and related lattice problems. In *Proceedings of the 15th Symposium on the Theory of Computing (STOC 1983)*, 99–108. ACM, 1983.
- [Kne57] M. Kneser. Klassenzahlen indefiniter quadratischer Formen in drei oder mehr Veränderlichen. *Archiv Math.*, 8:241–250, 1957.
- [LLL82] Arjen K. Lenstra, Hendrik W. Lenstra, and László Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261:515–534, 1982.
- [MG02] Daniele Micciancio and Shafi Goldwasser. *Complexity of lattice problems: a cryptographic perspective*. Volume 671 of The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, 2002. ISBN 0-7923-7688-9.
- [NS01a] G. Nebe and N.J.A. Sloane. The Catalogue of Lattices. http://www.research.att.com/∼njas/lattices/, 2001.
- [NS01b] Gabriele Nebe and Neil J.A. Sloane. A Catalogue of Lattices. http://akpublic.research.att.com/∼njas/lattices/index.html, 2001.
- [NS] P. Nguyen, D. Stehlé. “LLL on the Average.” 238–256.
- [NS09a] Phong Nguyen and Damien Stehlé. An LLL Algorithm with Quadratic Complexity. *SIAM Journal on Computing*, 39(3):874–903, 2009.
- [NS09c] Phong Q. Nguyen and Damien Stehlé. Low-dimensional lattice basis reduction revisited. *ACM Trans. Algorithms*, 5(4):Art. 46, 48, 2009. URL: https://doi.org/10.1145/1597036.1597050, doi:10.1145/1597036.1597050.
- [Pau98] Sachar Paulus. Lattice Basis Reduction in Function Fields. In *In ANTS-3 : Algorithmic*, 567–575. Springer-Verlag, 1998.
- [Poh87] Michael Pohst. A Modification of the LLL Reduction Algorithm. *J. Symbolic Comp.*, 4(1):123–127, 1987.

- [Pro] The SPACES Project. MPFR, a LGPL-library for multiple-precision floating-point computations with exact rounding. http://www.mpfr.org/.
- [PS08] Xavier Pujol and Damien Stehlé. Rigorous and efficient short lattice vectors enumeration. In *Advances in Cryptology—AsiaCrypt 2008*, LNCS. Springer, 2008.
- [SE94] Claus-Peter Schnorr and Michael Euchner. Lattice Basis Reduction: Improved Practical Algorithms and Solving Subset Sum Problems. *Mathematics of Programming*, 66:181–199, 1994.
- [SH95] Claus-Peter Schnorr and Horst Helmut Hörner. Attacking the Chor-Rivest Cryptosystem by Improved Lattice Reduction. In *Advances in Cryptology—EuroCrypt 1995*, volume 921 of LNCS, 1–12. Springer-Verlag, 1995.
- [SP91] Rainer Schulze-Pillot. An algorithm for computing genera of ternary and quaternary quadratic forms. In Stephen M. Watt, editor, *Proceedings ISSAC'91*, 134–143. Bonn, 1991.
- [Sho] Victor Shoup. NTL, Number Theory C++ Library. http://www.shoup.net/ntl/.
- [Sim05b] Denis Simon. Solving quadratic equations using reduced unimodular quadratic forms. *Math. Comp.*, 74(251):1531–1543 (electronic), 2005.
- [Ste09] Damien Stehlé. *Floating-point LLL: theoretical and practical aspects*. Springer-Verlag, 2009, To appear.
- [vEB81] Peter van Emde Boas. Another NP-complete partition problem and the complexity of computing short vectors in a lattice. Technical report 81-04, Mathematisch Instituut, Universiteit van Amsterdam, 1981.

Chapter 32

Lattices over Number Fields

32.1 Introduction

Some initial machinery for lattices over number fields appear in Magma in V2.22. In particular, automorphism group and isometries can be computed for totally positive definite lattices (over totally real fields). This work is inspired by code of Gael Collinet, and in the future will be subsumed into his project to implement functionality for Lorentz lattices.

The category of number field lattices is `LatNF`, and the elements of such lattices are `LatNFElt`, with these being user-defined types.

32.2 Number Field Lattices

32.2.1 Creation of Number Field Lattices

A lattice L over a number field K is a torsion-free $\mathbb{Z}_K$-module contained in K^n, together with an inner product having image in K. Note that the inner product is not required to be positive definite. Here n is the degree (of the vector space), and the rank m of the module is referred to as the rank or dimension of the lattice. A lattice with $m = n$ is called a *full* lattice.

Every such lattice admits a pseudobasis, which is an independent sequence of vectors $\vec{b}_i$ and a sequence of nonzero fractional ideals J_i such that every element in L can be written as $\sum_i a_i \vec{b}_i$ where $a_i \in J_i$. A **simple** lattice is one for which all the $J_i = (1)$, and lattices over principal ideal domains are always free (and can be made simple by rescaling).

A number field lattice L in Magma is specified by providing: a sequence of vectors $S = v_1, \ldots, v_m$; an optional sequence of fractional ideals $J_1, \ldots, J_m$; an optional inner product matrix M, which can be given either on K^n or as Gram matrix on the given vectors, namely $(v, w) = vM\overline{w}^{tr}$; and an optional involution $\cdot \mapsto \bar{\cdot}$ with respect to which the inner product is Hermitian, that is $(v, w) = \overline{(w, v)}$.

The Handbook chapter Modules over Dedekind Domains (Section Modules over Dedekind Domains), is closely related and functionality exists for passing to and from such modules and number field lattices.

`NumberFieldLattice(K, d): FldNum, RngIntElt -> LatNF`

```
Gram      : Mtrx                    Default: IdentityMatrix(K,d)
Involution: FldAut                  Default: IdentityAutomorphism(K)
```

Given a number field K and a degree d, create the standard lattice (with the identity as the basis). The `Gram` parameter (which must be invertible) can be used to specify the bilinear form on the basis vectors. The `Involution` parameter can be used to specify a field automorphism with respect to which the form is Hermitian.

`NumberFieldLattice(S): [ModTupFldElt] -> LatNF`

```
InnerProduct: Mtrx                      Default: 0
Gram        : Mtrx                      Default: 0
Ideals      : SeqEnum                   Default: []
Independent : BoolElt                   Default: false
Involution  : FldAut                    Default: 1
```

Given a sequence of vectors S over a number field, return the number field lattice determined by them. The inner product matrix on the ambient space may be given, or alternatively a Gram matrix on the given vectors (a check is made if this is consistent if the vectors are dependent). In either case, the pseudoGram matrix on the resulting pseudobasis must be invertible. If neither is given, the trivial (identity) inner product is assumed. The vectors can be dependent, unless the `Independent` parameter is set to `true`. The bilinear form can be set to be Hermitian with respect to a given involution on the base field, specified by `Involution`.

The sequence of ideals can be used to specify a non-simple lattice, as indicated in the Introduction above.

Note that Magma attempts to retain the given basis vectors (unless they are dependent), and does not apply reduction (either echelonisation or LLL-reduction), as such algorithms may not be available.

`NumberFieldLattice(D): ModDed -> LatNF`

```
Involution: FldAut                      Default: 1
```

Given a Dedekind module D over a number field, return the associated number field lattice, obtained by taking the `PseudoMatrix` containing its pseudobasis and fractional ideals. The module must not have a denominator (essentially meaning that it was not created by a quotient construction), so that it will be torsion-free. If the resulting lattice is free, it will be written in simple form. The bilinear form can be set to be Hermitian with respect to a given involution on the base field, specified by `Involution`.

`NumberFieldLattice(L): Lat -> LatNF`

Convert a lattice to a `LatNF` over the rationals as a number field.

`NumberFieldLatticeWithGram(F): Mtrx -> LatNF`

```
Involution: FldAut                      Default: 1
```

Given a regular matrix F over some number field K, which is Hermitian with respect to the involution `Involution`, create the free standard lattice (with the identity matrix as basis matrix) and Gram matrix F.

`StandardLattice(V): SpcPlr -> LatNF`

`StandardLattice(V): SpcPlr -> Lat`

The standard lattice in the polar space V.

`LatticeWithBasis(V, B): SpcPlr, Mtrx[RngOrd] -> LatNF`

`LatticeWithBasis(V, B): SpcPlr, Mtrx[FldAlg] -> LatNF`

`LatticeWithBasis(V, B): SpcPlr, Mtrx[RngInt] -> Lat`

`LatticeWithBasis(V, B): SpcPlr, Mtrx[FldRat] -> Lat`

`LatticeWithBasis(V, B, J): SpcPlr, AlgMatElt[RngOrd], [ RngOrdFracIdl ] -> LatNF`

`LatticeWithBasis(V, B, J): SpcPlr, AlgMatElt[FldAlg], [ RngOrdFracIdl ] -> LatNF`

`LatticeWithBasis(V, B, J): SpcPlr, AlgMatElt[FldQuad], [ RngOrdFracIdl ] -> LatNF`

`LatticeWithBasis(V, B, J): SpcPlr, AlgMatElt[RngInt], [ RngIntFracIdl ] -> Lat`

`LatticeWithBasis(V, B, J): SpcPlr, AlgMatElt[RngInt], [ RngInt ] -> Lat`

`LatticeWithBasis(V, B, J): SpcPlr, AlgMatElt[FldRat], [ RngIntFracIdl ] -> Lat`

LatticeWithBasis(V, B, J): SpcPlr, AlgMatElt[FldRat], [RngInt] -> Lat

Lattice in V with basis specified by the rows of the matrix B, and coefficient ideals J.

LatticeWithPseudobasis(V, P): SpcPlr, PMat -> LatNF

A lattice L in V with pseudo-matrix P.

Lattice(V, L): SpcPlr, ModDed -> LatNF

Lattice(V, L): SpcPlr, Lat -> Lat

The lattice in V with underlying Dedekind module L.

Module(L): LatNF -> ModDed

IP: BoolElt	Default: true

The number field lattice L is turned into a Dedekind module, by taking as a pseudomatrix the lattice pseudobasis and the coefficient ideals. This requires the ambient inner product to be known, unless the IP parameter is false (when the resulting module has the trivial inner product on the ambient).

ChangeRing(L, R): LatNF, Rng -> LatNF

The lattice $L \otimes R$ base-changed to R.

sub<L | RHS>: LatNF, Any -> LatNF, Map

Construct the sublattice S of a number field lattice L generated by RHS. The right hand side can consist of (sets, sequences, tuples, and lists of) number field lattices, Dedekind modules, or elements thereof; provided the elements (or their generators) can be coerced into L. The function also returns the inclusion map from S to L.

ext<L | RHS>: LatNF, Any -> LatNF, Map

Construct the lattice E generated by the number field lattice L and RHS. The right hand side can consist of (sets, sequences, tuples, and lists of) number field lattices, Dedekind modules, or elements thereof; provided the elements (or their generators) can be coerced into the embedding space of L. The function also returns the inclusion map from L to X.

A + B: LatNF, LatNF -> LatNF

Given two number field lattices A and B of the same degree over the same base ring, determine their join. If the pseudobasis vectors for A and B are independent, then the new lattice will have this union for its pseudobasis. The lattices must have the same inner product on the ambient space, and it must be known unless the K-span of either A and B contains the other (when the pseudoGram matrices are checked for consistency).

A meet B: LatNF, LatNF -> LatNF

Given two number field lattices A and B having the same degree over the same base ring, determine their intersection. The lattices must have the same inner product (if known) on the ambient space. If one lattice is contained in the other, then it will be returned in its original form.

r * L: RngElt, LatNF -> LatNF

L * r: Lat NF, LatNF -> LatNF

L / r: Lat NF, LatNF -> LatNF

BasisScaling(L, r): LatNF, RngElt -> LatNF

Given a number field lattice L and a nonzero scalar r coercible into the number field, return the lattice obtained by multiplying (or dividing) the basis vectors by the scalar.

InnerProductScaling(L, r): LatNF, RngElt -> LatNF

ScaledLattice(L, r): LatNF, FldElt -> LatNF

Given a number field lattice L and a nonzero scalar coercible r into the number field, return the lattice obtained by multiplying the inner product matrix (or Gram matrix if not available) by the scalar.

J * L: RngOrdFracIdl, LatNF -> LatNF

`L * J: LatNF, RngOrdFracIdl -> LatNF`

`L / J: LatNF, RngOrdFracIdl -> LatNF`

Given a number field lattice L and a nonzero fractional ideal J, return the number field lattice obtained by multiplying/dividing all the coefficient ideals by the given ideal.

`T * L: Mtrx, LatNF -> LatNF`

Given an invertible transformation matrix T of the same dimension as the given number field lattice L, return the lattice obtained by the given basis transformation. In other words, return the lattice $\{\sum_i a_i T\vec{b_i} : a_i \in J_i\}$. Note that T does not need to be integral.

`TJ * L: PMat, LatNF -> LatNF`

Given a pseudomatrix TJ with invertible transformation matrix T and nonzero coefficient ideals J, return the number field lattice obtained from number field lattice L by applying T to `L`s` pseudobasis and multiplying the coefficient ideals I_i of L accordingly. In other words the new lattice is $\{\sum_i a_i T\vec{b_i} : a_i \in I_i J_i\}$. Note that T does not need to be integral.

`L * T: LatNF, Mtrx -> LatNF`

Given an invertible transformation matrix T of the same degree as the given number field lattice L, return the lattice obtained by the given basis transformation on the ambient space. If the transformation does not map the pseudobasis span onto itself, the lattice must have an ambient inner product. Note that T does not need to be integral.

`DirectSum(A, B): LatNF, LatNF -> LatNF`

Given two number field lattices A and B over the same base ring, determine their direct sum. The underlying basis vectors are retained.

`DirectSum(A): SeqEnum[LatNF] -> LatNF`

Given a nonempty sequence of number field lattices $A_1, \ldots, A_n$ over the same base ring, determine their direct sum. The underlying basis vectors are retained.

`OrthogonalComplement(L, v): LatNF, LatNFElt -> LatNF`

`OrthogonalComplement(L, S): LatNF, LatNF -> LatNF`

Given two number field lattices $S \subset L$, or a vector v in a lattice L, determine the orthogonal complement.

`Dual(L): LatNF -> LatNF`

Given a number field lattice L, return its dual, given by the pseudobasis $G^{-1}B$ where B and G are the pseudobasis and pseudoGram matrix for L, with coefficient ideals $\overline{J}_i^{-1}$ for all i. The resulting pseudoGram matrix will be G^{-1}, while the inner product (if known) is preserved.

`SimpleLattice(L): LatNF -> LatNF`

Given a number field lattice L all of whose coefficient ideals are principal, return the lattice obtained by multiplying the basis vectors by generators. Alternatively, if the lattice is free but not already in principal form, compute the `SteinitzForm` of the Dedekind module to get principal coefficient ideals, and then proceed as above. There will always be an ambiguity with choosing generators of the principal ideals.

`ZLattice(L): LatNF -> Lat`

`RestrictionOfScalarsToZ(L): LatNF -> Lat`

Given a number field lattice L, return its restriction of scalars to $\mathbb{Z}$.

`MaximalSublattices(L, p): LatNF, RngOrdIdl -> [LatNF], [RngIntElt]`

`MinimalSuperlattices(L, p): LatNF, RngOrdIdl -> [LatNF], [RngIntElt]`

```
Limit     : RngIntElt                    Default: Infinity()
AutoOrbits: BoolElt                      Default: false
CallBack  : BoolElt                      Default: false
```

Given a number field lattice L and a prime ideal p, compute all sublattices (superlattices) of L of index p.

If `Limit` is set to some integer, the enumeration stops once `Limit` lattices have been found.

`AutoOrbits` can be set to a subgroup G of the automorphism group of L. Then only orbit representatives of the action under G are returned. A second return value then lists the sizes of the orbits. If L is totally positive definite, one can also set `AutoOrbits` to `true`. In this case G is taken to be the full automorphism group of L. Note that the function does not check whether G is a subgroup of the automorphism group of L.

The optional argument `CallBack` can be set to a function f which takes two arguments X and S and returns two booleans k and c. For each sub-/superlattice S, the above intrinsics then call f with the list X of the previously found lattices and the lattice S. If k is `true`, the lattice S is kept, i.e. appended to X. If c is `true`, the intrinsics continue, otherwise the enumeration stops immediately. (Note that the enumeration still stops once the list X contains `Limit` lattices.)

`MaximalIntegralLattice(Q): Mtrx -> LatNF`

Given a non-degenerate symmetric matrix, the function returns a lattice L with inner product matrix Q which is even and integral, and L is maximal with respect to this property.

`MaximalIntegralLattice(L): LatNF -> LatNF`

`MaximalIntegralLattice(L): Lat -> Lat`

`MaximalIntegralLattice(L, p): LatNF, RngOrdIdl -> LatNF`

`MaximalIntegralLattice(L, p): Lat, RngInt -> LatNF`

`MaximalIntegralLattice(L, p): Lat, RngIntElt -> LatNF`

```
Even: BoolElt                    Default: false
```

Given an integral number field lattice L, compute some maximal integral overlattice M of L. If `Even` is `true`, computes a maximal even overlattice M.

If a prime ideal p is given, then the resulting lattice M agrees with L at all places different from p and M_p is maximal integral, or maximal even integral, depending on `Even`.

`MaximalIntegralLattice(V): SpcPlr -> LatNF`

```
Even: BoolElt                    Default: false
```

Given a polar space V, constructs a maximal integral lattice in V. If `Even` is `true`, constructs a maximal even such lattice.

Example: Creating Number Field Lattices (ex-d20734)

We give some examples of lattice creation.

```
> K<s5> := NumberField(Polynomial([-5,0,1])); // Q(sqrt(5))
> L1 := NumberFieldLattice(K,3);
> G := Matrix(3,3,[K | 1,2,3, 2,s5,-1, 3,-1,0]);
> L2 := NumberFieldLattice(K,3 : Gram:=G); // with Gram matrix
> v1 := L1![1,2,3];
> v2 := L1![K.1,-1,2];
> L3 := NumberFieldLattice([Vector(v1),Vector(v2)]); // on vectors
> L4 := sub<L1|[v1,v2]>; // on LatNFElt's, same as L3
> M := Module(L3); // create the Dedekind module
> L5 := NumberFieldLattice(M); // and back to the NF lattice
> D := Dual(L3);
> L6 := D+L3;
> S := L3/K.1; // scaling
> L7 := S meet D; L7;
```

(continues on next page)

(continued from previous page)

```
Number field lattice over Number Field with defining polynomial x^2 - 5
over the Rational Field with basis matrix
[-3*s5 + 2         7         0]
[  -s5 + 1         3         1]
> T3 := Matrix(3,3,[K | 1,2,3, 4,5,6, K.1,-1,-2]);
> L8 := L7*T3; // transform is same degree, operate on right
> T2 := Matrix(2,2,[K | 1,2, K.1,-1]);
> L9 := T2*L7; // lattice is 2-dim, operate on left (pseudobasis)
> assert Dimension(L9) eq 2 and Degree(L9) eq 3;
> DS := DirectSum(L2,L9); // 5-dimensional
> assert Dimension(DS) eq 3+2;
> O1 := OrthogonalComplement(L1,L3);
> O2 := OrthogonalComplement(L1,v1);
> O1;
Number field lattice over Number Field with defining polynomial x^2 - 5
over the Rational Field with basis matrix
[  1/2*(7*s5 + 7)    1/2*(s5 + 13) 1/2*(-3*s5 - 11)]
> O2;
Number field lattice over Number Field with defining polynomial x^2 - 5
over the Rational Field with basis matrix
[       -s5 - 1   1/2*(s5 + 1)              0]
[1/2*(3*s5 + 3)              0  1/2*(-s5 - 1)]
```

32.2.2 Attributes of Number Field Lattices

`Basis(L): LatNF -> [ModTupFldElt]`

`BasisMatrix(L): LatNF -> Mtrx`

Given the simple number field lattice L, return the basis, either as a sequence of vectors, or as a row matrix.

`FreeBasis(L): LatNF -> [ModTupFldElt]`

Given a free lattice L, returns a basis for L.

`LocalBasis(L, p): LatNF, RngOrdIdl -> [ ModTupFldElt ]`

`LocalBasis(L, p): Lat, RngInt -> [ ModTupFldElt ]`

```
Type: MonStgElt                    Default: ""
```

Returns a basis of a free lattice F in the quadratic space generated by L such that F agrees with L at the completion at p. If `Type` is specified, it must either be `"Submodule"` or `"Supermodule"`. In which case F is either a sub- or supermodule of L.

`PseudoBasis(L): LatNF -> [ModTupFldElt]`

`PseudoBasisMatrix(L): LatNF -> Mtrx`

Given the number field lattice L, return the basis of its pseudobasis, either as a sequence of vectors, or as a row matrix.

`PseudoMatrix(L): LatNF -> PMat`

Given the number field lattice L, return the pseudomatrix of its coefficient ideals and pseudobasis.

`CoefficientIdeals(L): LatNF -> SeqEnum`

Given the number field lattice L, return its coefficient ideals.

`Involution(L): LatNF -> FldAut`

Given the number field lattice L, return the involution with respect to which it is sesquilinear.

`Generators(L): LatNF -> SeqEnum`

GeneratorMatrix(L): LatNF -> Mtrx

Given the number field lattice L, return a sequence of generators. If the lattice is simple, these are just the standard basis vectors.

InnerProductMatrix(L): LatNF -> Mtrx

Given the number field lattice L, return its inner product matrix. An error occurs if this is not known (for instance, when the Gram matrix was prescribed instead).

AmbientSpace(L): LatNF -> SpcPlr

The ambient polar space in which the number field lattice L embeds.

EmbeddingSpace(L): LatNF -> Mod

The embedding space of the number field lattice L.

MakeAmbientInnerProduct(~L, IP): LatNF

Given the number field lattice L having no ambient inner product, attach the given matrix IP as the inner product matrix for L. The matrix IP is checked to be consistent with the given pseudobasis and pseudoGram matrices.

GramMatrix(L): LatNF -> Mtrx

Given a simple number field lattice L, return its Gram matrix.

PseudoGramMatrix(L): LatNF -> Mtrx

Given a number field lattice L, return the Gram matrix of its pseudobasis.

GramMatrix(L, S): LatNF, [ModTupFldElt] -> AlgMatElt

GramMatrix(S): [LatNFElt] -> AlgMatElt

Half: BoolElt	Default: false

Given a number field lattice L and a sequence S of vectors in the lattice, return the Gram matrix of these vectors. If `Half` is set to `true`, scales the Gram matrix by a half.

Rank(L): LatNF -> RngIntElt

Dimension(L): LatNF -> RngIntElt

The rank (or dimension) of the number field lattice L.

Degree(L): LatNF -> RngIntElt

The degree (that is, dimension of the underlying ambient space) of the number field lattice L.

BaseRing(L): LatNF -> FldNum

The number field over which the number field lattice L is defined.

CoordinateRing(L): LatNF -> RngOrd

Order(L): LatNF -> RngOrd

The order $\mathbb{Z}_K$ over which the number field lattice L is a module.

Determinant(L): LatNF -> FldNumElt

The determinant of the Gram matrix of the simple number field lattice L. The determinant is well-defined up to squares of units on free lattices by first applying `SimpleLattice` if necessary.

Discriminant(L): LatNF -> RngOrdFracIdl

Volume(L): LatNF -> RngOrdFracIdl

The volume of the number field lattice L, which is the ideal generated by the determinants of all free sublattices. It is defined by $\det(G)\prod_i \mathrm{Nm}(J_i)$ where G is the pseudoGram matrix, J_i are the coefficient ideals, and $\mathrm{Nm}(I) = I\overline{I}$, where $\cdot \mapsto \bar{\cdot}$ is the involution.

When L is quadratic of odd rank, the discriminant returns half of this quantity (the half-discriminant).

`Norm(L): LatNF -> RngOrdFracIdl`

The norm of the lattice L, which is the ideal generated by the norms of all lattice elements. It equals $\sum_i \mathrm{Nm}(J_i)G_{ii} + \sum_{i,j} \mathrm{Tr}(J_i G_{ij} \overline{J}_j)$ where G is the pseudoGram matrix, the J_i are the coefficient ideals, and $\cdot \mapsto \bar{\cdot}$ is the involution.

`Scale(L): LatNF -> RngOrdFracIdl`

The scale of the lattice L, which is the ideal generated by the inner products of pairs of lattice elements. It equals $\sum_{i,j} J_i G_{ij} \overline{J}_j$, where G is the pseudoGram matrix, the J_i are the coefficient ideals, and $\cdot \mapsto \bar{\cdot}$ is the involution.

`BadPrimes(L): LatNF -> Set`

`Even: BoolElt`	`Default: false`

The set of all maximal ideals of the base ring of L at which L is not unimodular. If `Even` is set, then the primes over 2 are included.

`AuxiliaryForms(L): LatNF -> [ AlgMatElt ]`

Return a sequence of rational bilinear forms $\phi_1, \ldots, \phi_d$, defined by $\phi_i(x, y) = \mathrm{Tr}(a_i x, y)$, where a_i runs over a basis of the number field K.

`ElementaryDivisors(A, B): LatNF, LatNF -> [ RngOrdFracIdl ]`

The elementary divisors of the lattice A with respect to the lattice B. These are fractional ideals $D_1 \subseteq \ldots \subseteq D_n$ such that there exist pseudobases for A and B with the same basis vectors and coefficient ideals $J_{B,i} = D_i J_{A,i}$.

`Discriminant(A, B): LatNF, LatNF -> RngOrdFracIdl`

The discriminant of the lattice A with respect to the lattice B. This is the product of the elementary divisors of A with respect to B.

`Index(A, B): LatNF, LatNF -> RngOrdFracIdl`

The index of the lattice B in the lattice A.

`JordanDecomposition(L, p): LatNF, RndOrdIdl -> List, List, SeqEnum`

Given a number field lattice L over some field K and a prime ideal of $\mathbb{Z}_K$, a Jordan decomposition of L_p is an orthogonal decomposition of L_p into modular sublattices $L_1, \ldots, L_r$ such that the p-adic valuations of the scales of these sublattices are strictly increasing.

The first return value is a list of matrices $B_1, \ldots, B_r$ over K. The row spans of these lattices over $\mathbb{Z}_{K_p}$ yield a Jordan decomposition of L_p. The second and third return value list the Gram matrices and the p-adic valuations of the scales of the Jordan blocks respectively.

`MaximalNormSplitting(L, p): LatNF, RngOrdIdl -> SeqEnum, List`

A maximal norm splitting of a number field lattice L at some prime ideal p is an orthogonal decomposition of L_p into unary and binary sublattices $L_1, \ldots, L_r$ such that $\mathrm{scale}(L_i) \supseteq \mathrm{scale}(L_{i+1})$, $\mathrm{norm}(L_i) \supseteq \mathrm{norm}(L_{i+1})$, and $\mathrm{norm}(L_i^\#) \subseteq \mathrm{norm}(L_{i+1}^\#)$ for all $1 \leq i < r$. Here $L_i^\#$ denotes the dual of L_i.

`GoodBasisOfNormGenerators(L, p): LatNF, RngOrdIdl -> SeqEnum, SeqEnum`

Let L be a number field lattice and let p be some prime ideal. A basis of norm generators is a tuple of vectors $(x_1, \ldots, x_n)$ such that the norm of x_1 generates the norm of L_p and $(x_2, \ldots, x_n)$ is a basis of norm generators for $L \cap \langle x \rangle^\perp$. It is called good if the p-adic valuation of (x_i, x_i) is no larger then the valuation of (x_{i+2}, x_{i+2}) for all $1 \leq i \leq r-2$. Note that a basis of norm generators does not need to generate L_p but only a proper sublattice.

The function returns such a good basis of norm generators lying in KL as well as the norms of the basis vectors.

`GenusSymbol(L, p): LatNF, RngOrdIdl -> SeqEnum, Any`

`Uniformizer: RngElt`	`Default:`

Let $L_1, \ldots, L_r$ be a Jordan decomposition of L_p.

Suppose first p does not divide 2. The Genus symbol of L_p is a sequence of triples $< r_i, s_i, d_i >$ where r_i and s_i are the rank and the p-adic valuation of the scale of L_i. Further, for quadratic lattices $d_i = 1$ if and only if $\det(\pi^{-s_i} L_i)$ is a square in K_p^* for some uniformizer $\pi \in K$ of p, otherwise $d_i = -1$. For hermitian lattices, if the quadratic extension ramifies at p and s_i is even, $d_i = 1$ if and only if $\det(\pi^{-s_i} L_i)$ is a norm in K_p^*, and otherwise $d = -1$. When either s_i is odd, or the quadratic extension does not ramify, $d_i = 0$. The genus symbol only depends on the genus of L and π. Hence the function returns π as a second argument. One can also specify which uniformizer to choose with the optional argument `Uniformizer`.

If p divides 2, the functions returns the fundamental invariants of L_p, see [OMe73, IX.93] for quadratic lattices, and [Jac62a] for Hermitian lattices. It is a sequence of quadruples $< r_i, s_i, w_i, a_i >$ where r_i and s_i are as before. Further, for quadratic lattices, w_i and a_i denote the p-adic valuation of the weight and a norm generator of $L_i' = L_1 \perp \cdots \perp L_i$ respectively. For hermitian lattices, w_i denotes the p-adic valuation of the norm of L_i', and $a_i = det(L_i')$. Note that in this case, the genus symbol does not only depend on the genus of L, but also on the chosen Jordan decomposition. Hence the function returns the sequence of Gram matrices for the lattices L_i as a second argument.

`LocalGenus(L, p): LatNF, RngOrdIdl -> SymGenLoc`

`Uniformizer: RngElt`	`Default:`

The local genus symbol of L at p.

`Genus(L): LatNF -> SymGen`

`Proper: BoolElt`	`Default: false`

The genus symbol of L. If `Proper` is `true`, returns the proper genus.

`HasseInvariant(M, p): AlgMatElt, RngOrdIdl -> RngIntElt`

`Involution: Map`	`Default: 1`

Computes the Hasse invariant at the prime ideal p of the quadratic form given by the symmetric matrix M, whose base ring must be the order of p or its number field. If `Involution` is set to a field automorphism, the form is treated as the Hermitian form defined by M and that involution.

`HasseInvariant(L, p): LatNF, RngOrdIdl -> RngIntElt`

`AmbientSpace: BoolElt`	`Default: false`

Computes the Hasse invariant of the quadratic space V generated by L at the maximal ideal p. It is defined as a product of Hilbert symbols

$$\prod_{i<j} (a_i, a_j)_p$$

where $(a_1, \ldots, a_n)$ denotes some orthogonal basis of V.

If `AmbientSpace` is set, the function takes V to be the ambient space of the lattice L, which might have a larger rank than L.

`WittInvariant(M, p): AlgMatElt, RngOrdIdl -> RngIntElt`

`Involution: Map`	`Default: 1`

Computes the Witt invariant at the prime ideal p of the quadratic form given by the symmetric matrix M, whose base ring must be the order of p or its number field. If `Involution` is set to a field automorphism, the form is treated as the Hermitian form defined by M and that involution.

`WittInvariant(L, p): LatNF, RngOrdIdl -> RngIntElt`

`AmbientSpace: BoolElt`	`Default: false`

Let V be the quadratic space generated by L (or its ambient space if `AmbientSpace` is set). If the rank of V is odd (even) let C be the (even part of) the Clifford algebra of V. Then C is a tensor product of quaternion algebras over K_p and thus central simple. The Witt invariant of L_p is defined to be $+1$ is C is split and -1 otherwise.

`SpinorNorm(L, p): LatNF, RngOrdIdl -> ModTupFld, Map, BoolElt`

Let L be a definite number field lattice and let p be some prime ideal. The quotient $K_p^*/(K^*)^2$ is a vector space V over GF(2). Let $g\colon V \to K_p^*/(K^*)^2$ be an isomorphism of abelian groups. Let W be the subspace of V corresponding to the elements in $K_p^*/(K^*)^2$ that are spinor norms of proper isometries of L_p. The intrinsic returns W, g as well as a boolean that is `true` if and only if W corresponds to the subgroup $O_p^*(K_p^*)^2/(K_p^*)^2$. Note that the full space V can be recovered as `Domain`(g). The algorithm uses the results of M. Kneser [Kne57] and C. Beli [Bel03].

`Mass(L): LatNF -> FldRatElt`

`Mass(L): Lat -> FldRatElt`

`LocalFactor(L, p): LatNF, RngOrdIdl -> FldRatElt`

Let L be a definite number field lattice of rank m over an extension E of some totally real number field K of degree n, such that $[E : K] \leq 2$. Then the genus of L decomposes into finitely many isometry classes, represented by $L_1, \ldots, L_h$ say. The mass of L is defined to be

$$\mathrm{Mass}(L) = \sum_{i=1}^{h} \frac{1}{\#\mathrm{Aut}(L_i)}.$$

By Siegel's celebrated mass formula [Sie35] and its generalization [GHY01], the mass can be expressed as

$$\mathrm{Mass}(L) = 2^{-nr} L(M)\tau(G) \prod_p \lambda_p(L)$$

where the product runs over all maximal ideals p of the ring of integers of K, G is the isometry group (orthogonal or unitary), r is the rank of G, $\tau(G)$ is its Tamagawa measure, and $L(M)$ is a product of special values of L-functions. When $E = K$ (the quadratic case), this reads

$$\mathrm{Mass}(L) = 2^{-n\lfloor \frac{m}{2} \rfloor} \prod_{i=1}^{\lfloor \frac{m-1}{2} \rfloor} \zeta_K(1-2i) \cdot L(\chi, 1-m/2) \cdot \prod_p \lambda_p(L).$$

Here ζ_K denotes the usual Dedekind zeta function of K and $L(\chi, x)$ is only present if m is even, and in this case it is the L-series corresponding to χ. If the discriminant $d = (-1)^{m/2}\det(V)$ is a square, let $\chi = 1$. In all other cases, χ is the non-trivial character of $\mathrm{Gal}(K(\sqrt{d})/K)$. When E/K is a quadratic extension (the hermitian case), the formula reads

$$\mathrm{Mass}(L) = 2^{1-nm} \prod_{i=1}^{m} L(\chi^i, 1-i) \cdot \prod_p \lambda_p(L).$$

Here χ is the non-trivial character of $\mathrm{Gal}(E/K)$.

The above functions return the mass of L and the local factor $\lambda_p(L)$ respectively. The local factors are known in many but not all cases, for example:

- For odd prime ideals p, see W.T. Gan and J.K. Yu [GY00].
- For unramified even prime ideals p, see S. Cho [Cho15].
- If L_p is maximal, see G. Shimura [Shi99] or W.T. Gan, J. Hanke, and J.K. Yu [GHY01].
- For modular lattices, see M. Kirschmer [Kir16].

If none of the above results is applicable, a slow brute force search will be used to work out the local factors at the even prime ideals.

Neighbours(L, p): LatNF, RngOrdIdl -> [LatNF], [RngIntElt]

```
Limit     : RngIntElt                    Default: Infinity()
AutoOrbits: BoolElt                      Default: false
CallBack  : BoolElt                      Default: false
```

Given a number field lattice L and some prime ideal p such that L_p is modular, compute the p-neighbours of L i.e. the set of lattices M in the space generated by L such that $L \cap M$ has index p in both, M and L.

The automorphism group of L acts on L. If L is definite and `AutoOrbits` is set to `true`, the algorithm will only return orbit representatives of this action and the second return value gives the lengths of these orbits.

If `Limit` is set to some positive integer, the algorithm stops once it has found that many neighbours.

The optional argument `CallBack` can be set to a function f which takes two arguments (X, N) and returns two booleans (k, c). For each neighbour (or orbit representative) N the intrinsic encounters, it calls f with the list X of the previously found lattices and the lattice N. If k is `true`, the lattice N is kept, i.e. appended to X. If c is `true`, the intrinsic continues, otherwise the enumeration stops immediately. (Note that the enumeration still stops once the list X contains `Limit` lattices.)

IteratedNeighbours(L, p): LatNF, RngOrdIdl -> [LatNF]

```
Limit  : RngIntElt                   Default: Infinity()
UseAuto: BoolElt                     Default: true
```

Computes a system of representatives of the isometry classes of all lattices that can be reached from a definite number field lattice L by repeatedly applying the p-neighbour construction. The lattice L_p must be modular.

If `UseAuto` is set, the algorithm will make use of the automorphism groups of the lattices it encounters.

If `Limit` is set to some positive integer, the algorithm stops once it has found that many isometry classes.

GenusRepresentatives(L): LatNF -> [LatNF], Assoc

```
NaturalAction: BoolElt                   Default: true
Proper       : BoolElt                   Default: false
UseAuto      : BoolElt                   Default: true
ThetaPrec    : RngIntElt                 Default: 25
BadPrimes    : BoolElt                   Default: true
Limit        : RngIntElt                 Default: Infinity()
UseMass      : BoolElt                   Default: true
```

Returns a sequence of representatives of the isometry classes in the genus of the number field lattice L. If the lattice is definite, also returns an associative array whose keys are isometry invariants of the lattice (theta series with precision `ThetaPrec`), and values the index in the sequence of genus representatives. The lattice L is supposed to have rank at least 3. If `NaturalAction` is set to `true`, the representatives returned lie in the same polar space as L. If `Proper` is set to `true`, returns representatives for proper isometry classes. If `UseMass` is set to `false`, does not use the mass formula as a stopping condition, and compute the entire neighbour closure. If `BadPrimes` is set to `false`, does not iterate over neighbors at bad primes. If `Limit` is set to some integer, the algorithm stops once it has enumerated that many lattices. In the definite case, the algorithm will make use of the automorphism groups of the representatives, unless `UseAuto` is set to `false`.

The algorithm consists of two steps:

1. Enumerate all isometry classes in the spinor genus of L. For indefinite lattices, this step becomes void since spinor genus and isometry classes coincide thanks to Strong Approximation. In the definite case, iterated neighbours are computed at some suitable small prime ideal.

2. Enumerate the spinor genera in the genus of L. This step is well understood in theory, see for example [OMe73, X.102]. Using a computationally more accessible variation of this description [Kir16, Lor16], the algorithm then chooses several neighbours that are known to switch spinor genera.

Example: Attributes of Number Field Lattices (ex-58cb4b)

Here are examples of getting attributes of number field lattices.

```
> K := NumberField(Polynomial([5,0,1])); // Q(sqrt(-5))
> O := Integers(K);
> p2 := Factorization(2*O)[1][1]; // nonprincipal
> Js := [(2*O)/p2,2*O];
> v1 := Vector([K!1,0,0]);
> v2 := Vector([K!0,K.1,0]);
> L1 := NumberFieldLattice([v1,v2] : Ideals:=Js);
> assert not IsFree(L1); // not free
> assert not IsSimple(L1); // and not simple
> PseudoBasisMatrix(L1); // BasisMatrix does not work
[  1   0   0]
[  0 K.1   0]
> PseudoGramMatrix(L1);
[ 1  0]
[ 0 -5]
> CoefficientIdeals(L1);
[ Ideal of O Basis: [1 1]
                    [0 2],
  Principal Ideal of O, Generator: [2, 0] ]
> Generators(L1); // three of these, though L1 has dimension 2
[
    ( -3*$.1 - 7 -2*$.1 + 50           0),
    (  44*$.1 + 1100 1770*$.1 - 4130               0),
    ( 176*$.1 + 1628 2242*$.1 - 7080               0)
]
> assert #$1 eq 3 and Dimension(L1) eq 2;
> assert Degree(L1) eq 3 and Rank(L1) eq 2;
> assert BaseRing(L1) eq K;
> Discriminant(L1); // same as Volume
Principal Ideal Generator: [-40, 0]
> Norm(K.1*L1);
Principal Ideal of O, Generator: [10, 0]
> // now we take the direct sum of L1 with itself
> // which is free, and thus we can use other intrinsics
> D := DirectSum([L1,L1]);
> assert IsFree(D); // D itself is not in simple form
> Determinant(D); // works, as D is free
1600
> S := SimpleLattice(D);
> BasisMatrix(S); // works, while for D would not
[-3*K.1 - 5 -4*K.1 + 20 0 0 0 0]
[12*K.1 - 84 -100*K.1 - 20 0 5*K.1 - 1 0 0]
[-72*K.1 + 54 100*K.1 + 370 0 -7*K.1 - 17 -8*K.1 + 50 0]
[1080*K.1 - 864 -1560*K.1 - 5520 0 108*K.1 + 252 114*K.1 - 760 0]
> assert #Generators(S) eq 4; // same as the basis vectors
```

NumberOfIsotropicSubspaces(L, P, k): LatNF, RngOrdIdl, RngIntElt -> RngIntElt

NumberOfIsotropicSubspaces(L, P, k): Lat, RngInt, RngIntElt -> RngIntElt

NumberOfIsotropicSubspaces(L, P, k): Lat, RngIntElt, RngIntElt -> RngIntElt

The number of k-dimensional isotropic subspaces of L_P.

`NumberOfNeighbors(L, P, k): LatNF, RngOrdIdl, RngIntElt -> RngIntElt`

`NumberOfNeighbors(L, P, k): Lat, RngInt, RngIntElt -> RngIntElt`

`NumberOfNeighbors(L, P, k): Lat, RngIntElt, RngIntElt -> RngIntElt`

The number of P^k-neighbour lattices for L.

`NeighborProcess(L, P, k): LatNF, RngOrdIdl, RngIntElt -> NeighborProc`

`NeighborProcess(L, P, k): Lat, RngInt, RngIntElt -> NeighborProc`

```
Perestroika: BoolElt                    Default: false
```

A process for enumerating over P^k-neighbors of L. If `Perestroika` is set to `true`, enumerates the neighbors in the sense of the Perestroika operators.

`Neighbor(nProc): NeighborProc -> LatNF`

```
UseLLL     : BoolElt                    Default: false
Perestroika: BoolElt                    Default: false
```

A lattice which is the next neighbour in the process `nProc`. If `UseLLL` is `true`, applies `LLL` to find a representative with a small Gram matrix. If `Perestroika` is set to `true`, returns a Perestroika neighbour.

`Advance(nProc): NeighborProc`

Advances the process to the next neighbour.

`SkipTo(nProc, V, S): NeighborProc, SeqEnum[ModTupFldElt[FldFin]], AlgMatElt[FldFin]`

Advances the process to the neighbour determined by the isotropic subspace spanned by the vectors in V and the skew-symmetric matrix S.

32.2.3 Predicates on Number Field Lattices

`IsSimple(L): LatNF -> BoolElt`

Returns `true` when all the coefficient ideals of number field lattice L are trivial.

`IsFree(L): LatNF -> BoolElt`

Returns `true` if the number field lattice L is free. Freeness is determined by determining whether the Steinitz class of the associated Dedekind module is principal.

`IsZero(L): LatNF -> BoolElt`

Checks if the rank of L is 0.

`IsFull(L): LatNF -> BoolElt`

Returns `true` if L has full rank.

`IsHermitian(L): LatNF -> BoolElt`

Return `true` if and only if L is Hermitian with respect to a nontrivial involution.

`IsQuadratic(L): LatNF -> BoolElt`

Return `true` if and only if L is a quadratic lattice.

`IsTotallyPositiveDefinite(L): LatNF -> BoolElt`

Returns `true` if the number field lattice L is totally positive definite (which necessarily includes the base ring being totally real).

`IsDefinite(L): LatNF -> BoolElt, RngOrdElt`

Returns `true` if the number field lattice L is definite and if so, and element a such that rescaling the inner prodcurt with a yields a totally positive definite lattice.

Signature(F): Mtrx[FldAlg] -> ModTupRngElt[RngInt], ModTupRngElt[RngInt], RngIntElt

Given a symmetric matrix F over a totally real field, return the signature of F, i.e. the number of positive and negative eigenvalues under each real embedding, as well as the number of zero eigenvalues.

A eq B: LatNF, LatNF -> BoolElt

A ne B: LatNF, LatNF -> BoolElt

Return **true** (**false**) if the two number field lattices A and B are equal (not equal). Two number field lattices are said to be equal if they have the same degree, have compatible inner products and/or pseudoGram matrices, and are subsets of each other.

IsIdentical(A, B): LatNF, LatNF -> BoolElt

Return **true** if the two number field lattices A and B are identical. Two number field lattices are said to be identical if they have the same degree, rank, basis matrix, Gram matrix, and inner product matrix if given.

IsSublattice(S, L): LatNF, LatNF -> BoolElt, Mtrx

S subset L: LatNF, LatNF -> BoolElt

Given two lattices S and L of the same degree over the same number field and with the same ambient inner product (if given), determine whether the first is a sublattice of the second.

IsMaximal(L): LatNF -> BoolElt, LatNF

IsMaximal(L, p): LatNF -> BoolElt, LatNF

Decide if L or L_p is maximal, i.e. no overlattice has the same norm. If not, a proper overlattice with the same norm is returned.

IsIntegral(L): LatNF -> BoolElt

IsIntegral(L): Lat -> BoolElt

IsIntegral(L, p): LatNF, RngOrdIdl -> BoolElt

IsIntegral(L, p): Lat, RngInt -> BoolElt

IsIntegral(L, p): Lat, RngIntElt -> BoolElt

Returns **true** if L or L_p is integral, i.e. $(v, w) \in \mathbb{Z}_K$ for all v, w in L or L_p.

IsEven(L): LatNF -> BoolElt

Returns **true** if and only if L is even, i.e. if and only if $(v, v) \in 2\mathbb{Z}_K$ for all $v \in L$.

IsMaximalIntegral(L): LatNF -> BoolElt, LatNF

IsMaximalIntegral(L): Lat -> BoolElt, Lat

IsMaximalIntegral(L, p): LatNF, RngOrdIdl -> BoolElt, LatNF

IsMaximalIntegral(L, p): Lat, RngInt -> BoolElt, LatNF

IsMaximalIntegral(L, p): Lat, RngIntElt -> BoolElt, Lat

Even: BoolElt	Default: false

Decide (1) if the norm of L or L_p is integral and (2) if the lattice is maximal. If condition (1) holds but (2) does not, then a proper overlattice with integral norm is also returned. If **Even** is **true** decides instead (1) if L or L_p is even and (2) if the lattice is maximal among even lattices.

IsModular(L): LatNF -> BoolElt, RngOrdFracIdl

Determine whether the **Dual** of the lattice L is a scaling of the original, and if so, return the scaling factor.

IsModular(L, p): LatNF, RngOrdIdl -> BoolElt, RngIntElt

Determine if L_p is modular, i.e. the dual of L_p is equal to a scaling of L_p. If so, the valuation of the scaling factor is also returned.

IsIsotropic(L, p): LatNF, RngOrdIdl -> BoolElt

`IsIsotropic(L, p): LatNF, PlcNumElt -> BoolElt`

```
AmbientSpace: BoolElt                    Default: false
```

Checks if the completion of L (or its ambient space) at p is isotropic.

`IsLocallyIsometric(L1, L2, p): LatNF, LatNF, RngOrdIdl -> BoolElt`

```
CheckIsometricSpaces: BoolElt                    Default: true
```

Check if the p-adic completions of L_1 and L_2 are isometric. The function first checks if the lattices generate isometric ambient spaces. If this is already known to be the case one can suppress this check by setting `CheckIsometricSpaces` to `false`.

`IsSameGenus(L1, L2): LatNF, LatNF -> BoolElt`

Check if the number field lattices L_1 and L_2 lie in the same genus, i.e. they are isometric locally everyhwere.

`IsRationallyEquivalent(L1, L2, p): LatNF, LatNF, RngOrdIdl -> BoolElt`

`IsRationallyEquivalent(L1, L2, p): LatNF, LatNF, PlcNumElt -> BoolElt`

```
AmbientSpace: BoolElt                    Default: false
```

Given two lattices over some number field K, check if the quadratic space generated by the lattices are isometric over the completion K_p.

If `AmbientSpace` is set, the function compares the ambient spaces of the lattices instead.

`IsRationallyEquivalent(L1, L2): LatNF, LatNF -> BoolElt`

```
AmbientSpace: BoolElt                    Default: false
```

Given two lattices over some number field K, check if the quadratic space generated by the lattices are isometric.

If `AmbientSpace` is set, the function compares the ambient spaces of the lattices instead.

32.2.4 Totally Positive Definite Lattices

There is some additional functionality for totally positive definite lattices (which are necessarily over totally real fields).

`AutomorphismGroup(L): LatNF -> GrpMat`

`ProperAutomorphismGroup(L): LatNF -> GrpMat`

```
NaturalAction: BoolElt                    Default: false
Check        : BoolElt                    Default: false
```

Given a number field lattice L, determine its automorphism group (returned a matrix group which stabilizes the Gram matrix). The proper automorphisms are the ones that also preserve orientation (have determinant 1). This relies on vector enumeration and can be expensive to compute.

If the `NaturalAction` parameter is set, then the automorphisms returned act on the ambient space. Otherwise they act on the pseudobasis. When the rank is less than the degree, the pseudobasis will be artificially extended to full rank, and the automorphisms will fix each vector in the extended part.

If `Check` is `true`, then Magma checks that each automorphism group generator maps L to itself. The `Rank` must be positive unless `NaturalAction` is `true`, in which case the `Degree` of the lattice must be nonzero.

`IsIsometric(A, B): LatNF, LatNF -> BoolElt, Mtrx`

`IsProperlyIsometric(A, B): LatNF, LatNF -> BoolElt, Mtrx`

NaturalAction: BoolElt	Default: false

Given two totally positive definite number field lattices A and B over the same (totally real) number field, determine if they are (properly) isometric. If so, also return a transformation matrix on pseudobases, unless `NaturalAction` is set, in which case the transformation is on the ambient. This function can be costly to evaluate, as it relies on vector enumeration.

`IsSimilar(A, B): LatNF, LatNF -> BoolElt, Mtrx, FldNumElt`

NaturalAction: BoolElt	Default: false

Given two definite number field lattices A and B over the same (totally real) number field, determine if they are similar. If so, the function also returns a transformation T and a scalar a such that T induces an isometry betweem A and `InnerProductScaling`(B, a). As in the previous intrinsic, `NaturalAction` decides if T is with respect to the pseudobases of A and B or the standard bases of the ambient spaces.

`Sphere(L, e): LatNF, RngElt -> Setq`

Negatives: BoolElt	Default: true

Given a definite number field lattice L and a field element e, determine the set of vectors having that norm. Unless `Negatives` is `false`, the returned set contains both an element and its negation.

Examples of these are given in the final section of this chapter.

`IsRepresented(L, e): LatNF, RngElt -> BoolElt, LatNFElt`

Given a definite number field lattice L and a field element e, determine if there is a vector in L of norm e. If so, such a vector is returned as a second return value.

32.3 Number Field Lattice Elements

32.3.1 Creation

`Zero(L): LatNF -> LatNFElt`

The zero vector of the number field lattice L.

`L ! e: LatNF, Any -> LatNFElt`

Object e is coerced into the number field lattice L. The possibilities for the coerced object e are vectors of number field lattices, vectors in the proper degree ambient, and sequences of the proper length.

`L . i: LatNF, RngIntElt -> LatNFElt`

The ith pseudobasis vector of a number field lattice L. The given integer i must be nonnegative (0 gives the zero vector, also obtainable by `Zero`), and not exceed the rank of L. The ith coefficient ideal must also be trivial.

`CoordinatesToLattice(L, S): LatNF, SeqEnum -> LatNFElt`

`CoordinatesToLattice(L, v): LatNF, ModTupFldElt -> LatNFElt`

Given a sequence (or vector) S coercible into the coefficient field of number field lattice L whose length is equal to the rank of L, return the lattice vector with these coordinates. A check is made as to whether the vector (S or v) is in L.

32.3.2 Parent and Element Relations

`v in L: ModTupRngElt, LatNF -> BoolElt, ModTupFldElt`

`v in L: LatNFElt, LatNF -> BoolElt, ModTupFldElt`

Given a vector in an ambient space A of the number field lattice L where A has the same degree as L, determine whether v is in L. If so, then the coordinates of v with respect to the pseudobasis of L will also be returned. The coordinates of v will actually be returned whenever v lies in the K-span of the pseudobasis.

`Parent(v): LatNFElt -> LatNF`

The parent number field lattice to which the given lattice vector v belongs.

32.3.3 Arithmetic

`v + w: LatNFElt, LatNFElt -> LatNFElt`

`v - w: LatNFElt, LatNFElt -> LatNFElt`

`- v: LatNFElt -> LatNFElt`

`v eq w: LatNFElt, LatNFElt -> BoolElt`

`v ne w: LatNFElt, LatNFElt -> BoolElt`

`IsZero(v): LatNFElt -> BoolElt`

Addition, subtraction, negation, and (non)equality of the number field lattice elements v and w.

`s * v: RngElt, LatNFElt -> LatNFElt`

`v * s: LatNFElt, RngElt -> LatNFElt`

`v / s: LatNFElt, RngElt -> LatNFElt`

Given a vector v belonging to the number field lattice L defined over the number field K and an element s of K, scale v by s as indicated. The result is checked for membership of L.

`T * v: Mtrx, LatNFElt -> LatNFElt`

Given a vector v belonging to the number field lattice L defined over the number field K, and a matrix T defined over K, the pseudobasis coordinates of v are transformed by T. The result is checked for membership of L.

`v * T: LatNFElt, Mtrx -> LatNFElt`

Given a vector v belonging to the number field lattice L defined over the field K, and a matrix T acting on the ambient space of L, transform v by T. The result is checked for membership of L.

`v ^ M: LatNFElt, Mtrx -> LatNFElt`

Given an element v belonging to the number field lattice L and a matrix M acting on the ambient space of L, return the image of v under the transformation M. Here the action is on the coordinates of the vector (so M must be square, of dimensions equal to the rank of L), and the resulting vector must belong to the lattice.

`v ^ G: LatNFElt, GrpMat -> Setq[LatNFElt]`

`Orbit(G, v): GrpMat, LatNFElt -> Setq[LatNFElt]`

Given an element v belonging to the number field lattice L and a matrix group G acting on L, return the orbit of v under the action of G. This operation is also available if v is replaced by a set or sequence of elements of L. The user is responsible for ensuring that the orbit is finite.

`Stabilizer(G, v): GrpMat, LatNFElt -> GrpMat`

Given an element v belonging to the number field lattice L and a matrix group G acting on the coordinates of the vectors of L, return the stabilizer of v under the action of G. This operation is also available if v is replaced by a set or sequence of elements of L. The user is responsible for ensuring that the group G is finite.

`Norm(v): LatNFElt -> FldNumElt`

The norm of a given number field lattice element v.

`InnerProduct(v, w): LatNFElt, LatNFElt -> FldNumElt`

The inner product of two number field lattice elements v and w.

Example: Arithmetic of Number Field Lattice Elements (ex-9da2f4)

Here are some examples with number field lattice elements.

```
> K<s13> := NumberField(Polynomial([-13,0,1])); // Q(sqrt(13))
> L := NumberFieldLattice(K,3);
> v := Zero(L);
> assert IsZero(v);
> w1 := L.1;
> w2 := L.2-L.3;
> CoordinatesToLattice(L,Vector(5*w1-s13*w2));
(   5 -s13  s13)
> assert w2 in L;
> assert not Vector(w2)/2 in L; // cannot divide w2 by 2 directly
> assert Parent(v) eq L;
> Norm(w2);
2
> InnerProduct(w1,w2);
0
> T := Matrix(3,3,[K|s13,1,0, 3,-1,1+s13, s13,-s13,2+s13]);
> T*w2;
(-s13 + 3  s13 - 1       -1)
> w2*T; // same, as basis is standard
(-s13 + 3  s13 - 1       -1)
> S := sub<L|[w1,w2]>;
> Submatrix(T,1,1,2,2)*(S.1); // random input data, 2x2 mat in T*v
(s13   1  -1)
> G := AutomorphismGroup(L);
> assert #G eq 48;
> w2^G; // Orbit
{@
    ( 0  1 -1),
    (-1  1  0),
    ( 1  0 -1),
    (0 1 1),
    ( 1 -1  0),
    (-1  0 -1),
    (1 1 0),
    (1 0 1),
    ( 0 -1 -1),
    (-1  0  1),
    (-1 -1  0),
    ( 0 -1  1)
@}
> assert #$1 eq 12;
> #Stabilizer(G,w2); // 4*12 is 48
4
> #Stabilizer(G,w1);
8
> #Orbit(G,{w1,w2});
24
```

32.3.4 Access Functions

`Vector(v): LatNFElt -> ModTupFldElt`

Given an element v of the number field lattice L, return the underlying vector of the ambient space associated with v.

`Eltseq(v): LatNFElt -> SeqEnum`

Given an element v of the number field lattice L, return the sequence corresponding to the `Vector` of the element.

`Coordinates(v): LatNFElt -> ModTupFldElt`

Given an element v of the number field lattice L, return the coordinates of v, with respect to the pseudobasis of the parent lattice.

32.4 Examples

Example: Basic Number Field Lattices (ex-3a8daf)

Some basic functionality with number field lattices will be demonstrated. The standard lattice over $\mathbf{Q}(\sqrt{37})$ is created and a sublattice of it is selected.

```
> K<s37> := QuadraticField(37);
> L := NumberFieldLattice(K,3);
> r := (1+s37)/2;
> vecs:=[L.2+r*L.3, L.2-r*L.3, L.1+r*L.2];
> SUB := sub<L|vecs>;
```

The same sublattice may be obtained in another way, by directly giving its generating vectors as elements of the ambient space. Further, let S be the sublattice generated by one of the basis vectors of SUB. Then the lattice sum of S and its orthogonal complement O provide another sublattice.

```
> S2 := NumberFieldLattice([Vector(v) : v in vecs]);
> assert SUB eq S2; // both methods give the same
> S := sub<SUB|[vecs[1]]>; // 1-dimensional sublattice
> O := OrthogonalComplement(SUB, vecs[1]);
> InnerProduct(SUB!O.1, SUB!vecs[1]);
0
> InnerProduct(SUB!O.2, SUB!vecs[1]);
0
> assert IsSublattice(S+O, SUB);
> Norm(Determinant(S+O)/Determinant(SUB));
10201
> assert 101*SUB.2 in (S+O); // the index has norm 101^2
> assert 101*SUB.3 in (S+O);
```

Example: Gram Matrices and Transformations (ex-22b211)

The next example concerns non-trivial Gram matrices and transformations.

```
> K<s37> := QuadraticField(37);
> L := NumberFieldLattice(K,3); // standard lattice
> r := (1+s37)/2;
> v1 := Vector([1,2+3*r,-1+4*r]);
> v2 := Vector([1/2,1/3-3*r/2,1+r]);
> LAT := NumberFieldLattice([v1,v2]);
> Norm(Determinant(L+LAT));
1/1296
```

This lattice is 2-dimensional and has a denominator of 6. Next, while keeping the same basis vectors, the inner product matrix is changed. The resulting lattice is then modified by applying a random transformation matrix (not necessarily of unit determinant).

```
> IP := DiagonalMatrix([(s37-1)/2,1,1]);
> LATG := NumberFieldLattice([v1,v2] : InnerProduct:=IP);
```

(continues on next page)

(continued from previous page)

```
> R := Matrix(2,2, [s37,1, -1,(1+s37)/2]);
> NEW := R*LATG;
> BasisMatrix(NEW);
[     1/2*(2*s37 + 1)  1/12*(33*s37 + 661)    1/2*(3*s37 + 151)]
[       1/4*(s37 - 3) 1/12*(-25*s37 - 211)             -s37 + 9]
> InnerProductMatrix(NEW);
[1/2*(s37 - 1)             0             0]
[            0             1             0]
[            0             0             1]
> Determinant(R)^2*Determinant(LATG) eq Determinant(R*LATG);
true
```

If an integral transformation has determinant 1, then the resulting lattices are equal.

```
> Y := Matrix(3,3,[1,0,0, (1+s37)/2,1,0, -s37,-1,1]);
> Z := Matrix(3,3,[1,0,0, (-1+s37)/2,1,0, 2,s37,1]);
> TYZ := Y*Transpose(Z); // TYZ has determinant 1
> L eq TYZ*L;
true
> LYZ := NumberFieldLattice(K,3 : Gram:=TYZ*Transpose(TYZ));
> IsIsometric(TYZ*L,LYZ); // these have the same pseudoGram matrix
true
[1 0 0]
[0 1 0]
[0 0 1]
> assert PseudoGramMatrix(TYZ*L) eq PseudoGramMatrix(LYZ);
> C := Matrix(3,3,[1,s37,s37^2/2, 0,1,1+s37, 0,0,1]);
> D := Matrix(3,3,[1,(1+s37)/3,-1, 0,1,2+3*s37/4, 0,0,1]);
> TCD := C*Transpose(D);
> IsIsometric(L,TCD*L); // TCD has determinant 1, but is not integral
false
> assert Determinant(TCD) eq 1;
> [Denominator(x) : x in Eltseq(TCD)];
[ 6, 8, 2, 3, 4, 1, 1, 4, 1 ]
```

The next statement shows that the call to `IsIsometric` can be expensive in some cases.

```
> time IsIsometric(LAT,sub<LAT|[v1+v2,2*v1+v2]>); // nontrivial enum
Time: 27.960
true
[1 1]
[2 1]
```

Example: Lattices Related to the Icosahedron `(ex-dda13e)`

Here we work with lattices over $\mathbf{Q}(\sqrt{5})$ that are related to the icosahedron and dodecahedron.

```
> K<s5> := QuadraticField(5);
> L := NumberFieldLattice(K,3); // standard lattice
> phi := (1-s5)/2; // negative golden ratio
> vecs := [L.2+phi*L.3, L.2-phi*L.3, -L.1-phi*L.2];
> SUB := sub<L|vecs>;
> LAT := NumberFieldLattice([Vector(v) : v in vecs]);
> LAT eq SUB; // either way gives the same lattice
true
> Determinant(LAT); // (1-s5)^2
-2*s5 + 6
```

The given vectors are all of norm $(5-\sqrt{5})/2$, and their orbit under the automorphism group forms the vertices of a dodecahedron. The automorphism group of this sublattice is a central 2-extension of Alt(5), while that of the standard lattice is a central 2-extension of Sym(4).

```
> [Norm(v) : v in vecs];
[ 1/2*(-s5 + 5), 1/2*(-s5 + 5), 1/2*(-s5 + 5) ]
> A := AutomorphismGroup(LAT);
> IsIsomorphic(A/Center(A),AlternatingGroup(5));
true
> B := AutomorphismGroup(L);
> IsIsomorphic(B/Center(B),SymmetricGroup(4));
true
```

Carefully note that acting on vecs[1] as a member of L by a matrix is different to acting on vecs[1] as a member of LAT by the same matrix, as the action is on the coordinate vectors.

```
> vecs[1]^(A.1);
(1/2*(-s5 + 1) 1/2*(-s5 + 5)  1/2*(s5 - 3))
> Norm($1); // not the same
1/2*(-9*s5 + 25)
> (LAT!vecs[1])^(A.1);
(1/2*(s5 - 1)            0            1)
> Norm($1); // as desired
1/2*(-s5 + 5)
```

There are twelve vectors of norm $(5-\sqrt{5})/2$ in the smaller lattice, while there are 24 in the standard lattice.

```
> #Sphere(L,1); // 6 vectors of norm 1
6
> #Sphere(L,(5-s5)/2); // 24 vectors of this norm
24
> #Sphere(L,3); // 32 of norm 3
32
> assert Set(vecs) subset Set(Sphere(L,(5-s5)/2));
> #Sphere(LAT,1); // no vectors of norm 1
0
> #Sphere(LAT,(5-s5)/2); // 12 vectors of this norm
12
> #Sphere(LAT,3); // 20 of norm 3
20
> Lvecs := ChangeUniverse(vecs, LAT);
> assert Set(Lvecs) subset Set(Sphere(LAT,(5-s5)/2));
```

The orbit of the 12 vectors of norm $(5-\sqrt{5})/2$ in LAT is complete under the automorphism group. By taking a 3-sided face of the icosahedron, or a 5-sided face of the dodecahedron (vectors of norm 3), its action under the group automorphisms will give all such faces.

```
> Set(Sphere(LAT,(5-s5)/2)) eq Orbit(A,Lvecs[1]);
true
> FACE3 := {LAT.1,LAT.2,LAT.3}; // vertices of a face
> #(FACE3^A); // all 20 faces, each vertex appears 5 times
20
> Multiplicities(Multiset(&cat[SetToSequence(x) : x in FACE3^A]));
[ 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5 ]
> assert &and[Norm(InnerProduct(a,b)) eq -1 : a,b in FACE3 | a ne b];
> // now the dodecahedron
> S3 := Sphere(LAT,3);
```

(continues on next page)

(continued from previous page)

```
> w1 := LAT ! [(1+s5)/2,0,-(1-s5)/2];
> w2 := LAT ! [1,-1,-1];
> w3 := LAT ! [1,-1,1];
> w4 := LAT ! [(1+s5)/2,0,(1-s5)/2];
> w5 := LAT ! [-(1-s5)/2,-(1+s5)/2,0];
> FACE5 := {w1,w2,w3,w4,w5}; // vertices of a face
> #(FACE5^A); // all 12 faces, each vertex appears thrice
12
> Multiplicities(Multiset(&cat[SetToSequence(x) : x in FACE5^A]));
[ 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3 ]
> O1 := OrthogonalComplement(LAT,w1);
> O2 := OrthogonalComplement(LAT,w2);
> IsIsometric(O1,O2);
true
[      -s5 + 3   1/2*(s5 + 1)]
[  1/2*(s5 + 3) 1/2*(3*s5 + 7)]
```

Example: Nontrivial Coefficient Ideals `(ex-077d76)`

Here are some more examples with nontrivial coefficient ideals.

```
> K<u> := QuadraticField(257); // class number 3
> O := Integers(K);
> v1 := Vector([K|1,0,0]);
> v2 := Vector([K|0,u,0]);
> p2 := Factorization(2*O)[1][1]; // nonprincipal
> Js := [(2*O)/p2,2*O];
> L := NumberFieldLattice([v1,v2] : Ideals:=Js);
> IsFree(L);
false
> DS := DirectSum([L,L,L]); // Steinitz class is principal
> S := SimpleLattice(DS);
> DS eq S;
true
```

The automorphism group in the presence of nontrivial coefficient ideals need not have integral transforms on the pseudobasis.

```
> K<u> := QuadraticField(5);
> O := Integers(K);
> Vs := [Vector([K|2,0,1]),Vector([K|0,1,2])];
> Js := [1*O,2*O];
> G := DiagonalMatrix([K|4,1]);
> L := NumberFieldLattice(Vs : Gram:=G,Ideals:=Js);
> AutomorphismGroup(L);
MatrixGroup(2, K) of order 2^3
Generators:
    [ 1  0]    [-1  0]    [ 0   -2]
    [ 0 -1]    [ 0 -1]    [-1/2  0]
> A := AutomorphismGroup(L : NaturalAction);
> L*(&*Generators(A)) eq L;
true
```

The enumeration code with `Sphere` is also consistent with coefficient ideals, and the vectors are always returned in the ambient.

```
> Sphere(L,4);
{ (2 0 1), (-2  0 -1), (0 2 4), ( 0 -2 -4) }
> Sphere(L,8);
{ ( 2 -2 -3), (-2  2  3), (2 2 5), (-2 -2 -5) }
> Sphere(L,20 : Negatives:=false);
{ (2*u 0 u), ( 4 -2 -2), ( 2 -4 -7), (2 4 9), (4 2 6), (0 2*u 4*u) }
> [Coordinates(v) : v in $1]; // coordinates
[ (-u 0), ( 2 -2), ( 1 -4), (1 4), (2 2), (0 -2*u) ]
```

The `IsIsometric` functionality also works with coefficient ideals. This is demonstrated by first intrinsically transforming a lattice L by a unimodular transform matrix T to get a lattice N, and then finding the isometry between L and N on the ambient space.

```
> K<u> := QuadraticField(257);
> O := Integers(K);
> v1 := Vector([K|1,0,0]);
> v2 := Vector([K|0,u,0]);
> I := [1*O,2*O];
> G := Matrix(2,2,[K|1,0,0,2]);
> L := NumberFieldLattice([v1,v2] : Gram:=G,Ideals:=I);
> T := Matrix(2,2,[u-1,5,51,u+1]);
> PB :=PseudoBasis(L);
> M := NumberFieldLattice(PB : Gram:=T*G*Transpose(T),Ideals:=I);
> b,U := IsIsometric(T*L,M : NaturalAction); assert b;
> T*L*U^(-1) eq L;
true
```

Example: The Maass Unimodular Lattice (ex-4a3ec7)

Maass showed [Maa41] that there is exactly one even 4-dimensional unimodular lattice on $\mathbf{Z}[\phi]$ where $\phi = (1+\sqrt{5})/2$ is the golden ratio. The size of its automorphism group is 14400. There are 120 vectors of norm 2.

```
> K<u> := QuadraticField(5);
> e := (1+u)/2;
> M := Matrix(4,[2,-1,0,1-e, -1,2,-1,e-1, 0,-1,2,-e, 1-e,e-1,-e,2]);
> Determinant(M);
1
> L := NumberFieldLattice(K,4 : Gram:=M);
> Dual(L) eq L; // self-dual, since unimodular
true
> Norm(L); // L is even
Principal Ideal, Generator: 2
> AutomorphismGroup(L);
MatrixGroup(4, K) of order 2^6 * 3^2 * 5^2
> #Sphere(L,2);
120
```

It was shown by Costello and Hsia [CH87] that there is a unique rootless even 12-dimensional unimodular lattice on $\mathbf{Z}[\phi]$, with automorphism group of size 72576000. This is associated to a $\mathbf{Z}[\phi]$-structure on the Leech lattice.

To construct this lattice, it is first convenient to transform to coordinates where the hyperbolic decomposition over the prime ideal (2) is more perspicacious, so that the norms are multiples of 4, and the inner products are even except for the first/third and second/fourth vector pairs (this can be done by selecting some vectors of norm 4 that generate the lattice, and then performing suitable transformations). Moving then to the triple sum of the resulting lattice, an index 64 sublattice corresponding to the hyperbolic decomposition is found. Rescaling the inner product yields the desired unimodular lattice.

```
> H := Matrix(K,4, [     20,   -20*u+8,   u-6,       0,
>                   -20*u+8, -16*u+104, 6*u-8, (1-u)/2,
>                       u-6,     6*u-8,     4,       0,
>                         0,   (1-u)/2,     0,       4]);
> LL := NumberFieldLattice(K,4 : Gram:=H); // det(H) is e^4
> assert IsIsometric(L,LL);
> D := DirectSum([LL,LL,LL]);
> DIAG := [D.1+D.5+D.9, D.2+D.6+D.10];
> ORTHO := [D.3-D.7, D.3-D.11, D.4-D.8, D.4-D.12];
> assert &and[InnerProduct(a,b) eq 0 : a in DIAG, b in ORTHO];
> SUB := (2*D) + sub<D | DIAG cat ORTHO>;
> I6 := IdentityMatrix(K,6);
> T := DiagonalJoin(I6/e,I6); // make determinant be 1
> N := T*InnerProductScaling(SUB,1/2);
> Determinant(N);
1
> Dual(N) eq N; // again self-dual
true
> Norm(N); // and even
Principal Ideal, Generator: 2
> time #AutomorphismGroup(N);
72576000
Time: 1.150
> #Sphere(N,2); // rootless
0
> time #Sphere(N,4 : Negatives:=false);
18900
Time: 6.370
```

32.5 Lorentzian Lattices

Closely related to totally positive definite lattices are Lorentzian lattices. These are defined over totally real fields, and have exactly one embedding for which the lattice is not positive definite, at which it has a Lorentzian signature $(d-1,1)$ in dimension d.

A vector that has negative norm at this distinguished embedding is said to be timelike, while a vector with totally positive norm is spacelike.

32.5.1 Special Intrinsics

IsLorentzian(L): LatNF -> BoolElt, ModTupFldElt, RngIntElt

Given a number field lattice L over a totally real field, return `true` f L is Lorentzian and if so, also a return a lattice vector that has negative norm at the given embedding.

IsTimelike(v): LatNFElt -> BoolElt

IsSpacelike(v): LatNFElt -> BoolElt

Given a lattice field vector v in a Lorentzian lattice L return true if v is timelike (respectively spacelike). If the lattice is not Lorentzian, an error is signaled.

AutomorphismGroup(L, v): LatNF, LatNFElt -> GrpMat, GrpMat

NaturalAction: BoolElt	Default: false

Given a number field lattice L and a timelike vector v in it, determine the joint stabilizer. This is computed by determining the stabilizer (automorphism group) of the orthogonal complement of the vector and extending it to the lattice by requiring the vector also to be fixed. In other words, the group $A = \mathrm{Aut}(v \oplus v^{\perp})$ is computed. The subgroup of A that stabilizes the lattice is

then returned, while A is returned as a second value. If the `NaturalAction` parameter is set, the transformations are rewritten on the ambient space.

`IsIsometric(L, v, w): LatNF, LatNFElt, LatNFElt -> BoolElt, Mtrx`

```
NaturalAction: BoolElt                    Default: false
```

Given a number field lattice L and two timelike vectors v and w in it, determine whether there is an automorphism of L that sends w to v, and if so return such an isometry as the second argument. If the `NaturalAction` parameter is set, transformations are rewritten on the ambient space.

Example: Isometries of a Lorentzian Lattice `(ex-ac8528)`

We construct an easy Lorentzian lattice and compute isometries and automorphism group with various timelike vectors.

```
> K<u> := QuadraticField(5);
> G := DiagonalMatrix([-(u+1)/2,1,1,1]);
> L := NumberFieldLattice(K,4 : Gram:=G);
> IsLorentzian(L);
true (1 0 0 0) 1
> v := L![(u+1)/2,(u+1)/2,0,1];
> w := L![(u+1),(u+3)/2,(u+1)/2,(u+3)/2];
> assert IsTimelike(v) and IsTimelike(w);
> Norm(v), Norm(w);
1/2*(-u + 1)  1/2*(-u + 1)
> b, T := IsIsometric(L,v,w); assert b; T;
[       2*u + 5 1/2*(5*u + 11)          u + 2    1/2*(u + 1)]
[1/2*(-3*u - 5) 1/2*(-3*u - 7)   1/2*(-u - 3)   1/2*(-u - 1)]
[  1/2*(-u - 1)   1/2*(-u - 1)   1/2*(-u - 1)              0]
[        -u - 2         -u - 3   1/2*(-u - 1)              0]
> assert T*w eq v;
> #AutomorphismGroup(L,L![1,0,1,(3-u)/2]);
16
> #AutomorphismGroup(L,L![1,0,0,0]);
48
> #AutomorphismGroup(L,L![1,1,0,0]);
48
> #AutomorphismGroup(L,L![1,(u-1)/2,(u-1)/2,(u-1)/2]);
120
> s := L![1,0,(u-1)/2,(u-1)/2];
> A, B := AutomorphismGroup(L,s);
> #A,#B; // the automorphism group of s+O is larger than L
20 40
> O := OrthogonalComplement(L,s); // s+O has index 5 in L
> assert #AutomorphismGroup(O) eq #B;
> Norm(Determinant(sub<L|[s]>+O)/Determinant(L));
25
```

32.6 Bibliography

- [Bel03] C. N. Beli. Integral spinor norm groups over dyadic local fields. *J. Number Theory*, 102(1):125–182, 2003. URL: http://dx.doi.org/10.1016/S0022-314X(03)00057-X, doi:10.1016/S0022-314X(03)00057-X.
- [Cho15] S. Cho. Group schemes and local densities of quadratic lattices in residue characteristic 2. *Compos. Math.*, 151(5):793–827, 2015. URL: http://dx.doi.org/10.1112/S0010437X14007829, doi:10.1112/S0010437X14007829.

- [CH87] P. J. Costello and J. S. Hsia. Even Unimodular 12-Dimensional Quadratic Forms over $\mathbf{Q}(\sqrt{5})$. *Adv. Math.*, 64:241–278, 1987.
- [GY00] W. T. Gan and J.-K. Yu. Group schemes and local densities. *Duke Math. J.*, 105(3):497–524, 2000. URL: http://dx.doi.org/10.1215/S0012-7094-00-10535-2, doi:10.1215/S0012-7094-00-10535-2.
- [GHY01] Wee Teck Gan, Jonathan P. Hanke, and Jiu-Kang Yu. On an exact mass formula of Shimura. *Duke Math. J.*, 107(1):103–133, 2001. URL: http://dx.doi.org/10.1215/S0012-7094-01-10716-3, doi:10.1215/S0012-7094-01-10716-3.
- [Jac62a] Ronald Jacobowitz. Hermitian forms over local fields. *Amer. J. Math.*, 84:441–465, 1962. URL: https://doi.org/10.2307/2372982, doi:10.2307/2372982.
- [Kir16] M. Kirschmer. *Definite quadratic and hermitian form with small class number.* Habilitation, RWTH Aachen University, 2016.
- [Kne57] M. Kneser. Klassenzahlen indefiniter quadratischer Formen in drei oder mehr Veränderlichen. *Archiv Math.*, 8:241–250, 1957.
- [Lor16] D. Lorch. *Einklassige Geschlechter orthogonaler Gruppen.* PhD thesis, RWTH Aachen University, 2016.
- [Maa41] H. Maass. Modulformen und quadratische Formen über dem quadratischen Zahlkörper $R(\sqrt{5})$. *Math. Ann.*, 118:65–84, 1941.
- [OMe73] O. T. O'Meara. *Introduction to Quadratic Forms.* Springer, 1973.
- [Shi99] G. Shimura. An exact mass formula for orthogonal groups. *Duke Math. J.*, 97(1):1–66, 1999. URL: http://dx.doi.org/10.1215/S0012-7094-99-09701-6, doi:10.1215/S0012-7094-99-09701-6.
- [Sie35] C. L. Siegel. Über die analytische Theorie der quadratischen Formen. *Ann. of Math. (2)*, 36(3):527–606, 1935. URL: http://dx.doi.org/10.2307/1968644, doi:10.2307/1968644.

Chapter 33

Lattices with Group Action

33.1 Introduction

In Magma, a G-lattice L is a lattice upon which a finite integral matrix group G acts by right multiplication. Magma allows various computations with lattices associated with finite integral matrix groups by use of G-lattices. The computation of the automorphism group of a lattice (i.e. the largest matrix group that acts on the lattice) and the testing of lattices for isometry is performed within Magma by a search designed by Bill Unger, which is based on the Plesken-Souvignier backtrack algorithm [PS97a], together with ordered partition methods. Optionally, this may be combined with orthogonal decomposition code of Gabi Nebe. If G is a finite integral matrix group, then Magma uses Plesken's centering algorithm ([Ple74]) to construct all G-invariant sublattices of a given G-lattice L. The lattice of G-invariant sublattices of L can be explored much like the lattice of submodules over finite fields.

33.2 Automorphism Group and Isometry Testing

The functions in this section compute the automorphism group of a lattice and test lattices for isometry. Currently the lattices to which these functions are applied must be exact (over $\mathbb{Z}$ or $\mathbb{Q}$).

`AutomorphismGroup(L): Lat -> GrpMat`

`ProperAutomorphismGroup(L): Lat -> GrpMat`

```
Stabilizer   : RngIntElt                    Default: 0
BacherDepth  : RngIntElt                    Default: -1
Generators   : [ GrpMatElt ]                Default:
NaturalAction: Bool                         Default: false
Decomposition: Bool                         Default: false
Vectors      : Mtrx                         Default:
```

This function computes the automorphism group G of a lattice L which is defined to be the group of those automorphisms of the $\mathbb{Z}$-module underlying L which preserve the inner product of L. The proper automorphisms are the ones that also preserve orientation (have determinant 1). L must be an exact lattice (over $\mathbb{Z}$ or $\mathbb{Q}$). The group G is returned as an integral matrix group of degree m acting on the *coordinate* vectors of L by multiplication where m is the rank of L. The coordinate vectors of L are the elements of the coordinate lattice C of L which has the same Gram matrix as L, but standard basis of degree m (C can be created using the function `CoordinateLattice`). Since there is no natural matrix action of the automorphism group on L in the case that the rank of L is less than its degree, G does not act on the elements of L. However, if the rank of L equals its degree, then the parameter `NaturalAction` may be set to `true`, in which case the resulting group has the natural action on the basis vectors (not the coordinate vectors); note that in this case the resulting matrix group will have (non-integral) rational entries in general, even though the image under the group of an integral basis vector will always be integral.

The algorithm uses a backtrack search to find images of the basis vectors. A vector is a possible image if it has the correct inner product with the images chosen so far. Additional invariants which have to be respected by automorphisms are used by default and are usually sufficient for satisfactory performance. For difficult examples the parameters described below allow to consider further invariants which are more sophisticated and harder to compute, but often find dead ends in the backtrack at an early stage.

The algorithm computes and stores a set of short vectors in the lattice that spans the lattice and is guaranteed closed under the action of the automorphism group. This restricts its general application, as for high dimensional lattices the number of vectors of minimal length may be too large to work with. However, the function can of course be applied to lattices in higher dimensions with a reasonable number of short vectors.

Setting the parameter `Stabilizer := i` will cause the function to compute only the point stabilizer of i basis vectors. These will in general not be the first i basis vectors, as the function chooses the basis to speed up the computation.

The parameter `Depth` is retained for compatibility with previous versions of Magma (which used Souvignier's AUTO program) but it is now ignored. The improved backtrack search achieved by using ordered partition methods has made the `Depth` concept unnecessary.

In some hard examples one may want to use Bacher polynomials, which are a combinatorial invariant that usually separates the orbits of the automorphism group on the short vectors. However, these are expensive to calculate and should only be used if one suspects that the automorphism group has many orbits on the short vectors. The parameter `BacherDepth` specifies to which depth the Bacher polynomials may used and should usually be chosen to be 1, since even small automorphism groups will have only very few (most likely 1) orbits on the vectors having correct scalar product with the first image. Setting `BacherDepth` to zero forbids the use of Bacher polynomial invariant. Setting it to anything else allows the algorithm to use this invariant at level 1. Bacher polynomials are computed by counting pairs of vectors having a certain scalar product with other vectors. This scalar product is by default chosen to be half the norm of the vector, since this will usually be the value which occurs least frequent.

In some situations one may already know a subgroup of the full automorphism group, either by the construction of the lattice or an earlier stabilizer computation. This subgroup can be made available by setting `Generators := Q`, where Q is a set or sequence containing the generators of the subgroup.

Since V2.13, the algorithm has had the ability to first attempt to compute an orthogonal decomposition of L by considering the sublattices spanned by the set of shortest vectors, and if there is a non-trivial decomposition the automorphism group is computed via an algorithm of G. Nebe which computes the automorphism groups for the components and combines these. This decomposition method can be invoked by setting `Decomposition` to `true`.

When decomposition is suppressed, it is possible to supply the backtrack algorithm with the set of vectors to be used as domain for the search. To do this set the parameter `Vectors` to be a matrix so that the rows of the matrix give the lattice elements to be used. (Only one of a vector and its negative should be given.) To guarantee correctness of the result, the rows of the matrix should satisfy two conditions:

- The rows, together with their negations, must be closed under the action of the full automorphism group, and
- The sublattice of L generated by the rows of the matrix must equal L.

  These conditions are not checked by the code, and it is up to the user of this parameter to ensure the correctness of their input. For example, the function `ShortVectorsMatrix` returns a matrix which satisfies the first condition. If the first condition is not satisfied, there are no guarantees about what will happen. The second condition may be violated and still give a useful result. In particular, if the sublattice generated by the vectors given has finite index in the full lattice, then the final result will be the correct automorphism group. The problem is that the backtrack search may not be particularly efficient. This may still be better than working with a very large set of vectors satisfying the second condition.

AutomorphismGroup(L, F): Lat, [AlgMatElt] -> GrpMat
AutomorphismGroup(L, F): Lat, { AlgMatElt } -> GrpMat

```
Stabilizer : RngIntElt                          Default: 0
BacherDepth: RngIntElt                          Default: 0
Generators : [ GrpMatElt ]                      Default:
Vectors    : Mtrx                               Default:
```

This function computes the subgroup of the automorphism group of the lattice L which fixes also the forms given in the set or sequence F. The matrices in F are not required to be positive definite or even symmetric. This is for example useful to compute automorphism groups of lattices over algebraic number fields. The parameters are as above.

AutomorphismGroup(F): [AlgMatElt] -> GrpMat

```
Stabilizer : RngIntElt                          Default: 0
BacherDepth: RngIntElt                          Default: 0
Generators : [ GrpMatElt ]                      Default:
```

This function computes the matrix group fixing all forms given as matrices in the sequence F. The first form in F must be symmetric and positive definite, while the others are arbitrary. The call of this function is equivalent to `AutomorphismGroup( LatticeWithGram(F[1]), [ F[i] : i in [2..#F] ] )`. This function can be used to compute the Bravais group of a matrix group G which is defined to be the full automorphism group of the forms fixed by G. The parameters are as above.

Example: Automorphisms of E8 Acting on Vectors (ex-2d044f)

In this example we compute the automorphism group of the root lattice E_8 and manually transform the action on the coordinates into an action on the lattice vectors. Note that this exactly the same as using the `NaturalAction` parameter for the function `AutomorphismGroup`.

```
> L := Lattice("E", 8);
> G := AutomorphismGroup(L);
> #G; FactoredOrder(G);
696729600
[ <2, 14>, <3, 5>, <5, 2>, <7, 1> ]
> M := MatrixRing(Rationals(), 8);
> B := BasisMatrix(L);
> A := MatrixGroup<8, Rationals() | [B^-1 * M!G.i * B : i in [1 .. Ngens(G)]]>;
> A;
MatrixGroup(8, Rational Field)
Generators:
    [   0    0 -1/2  1/2 -1/2  1/2    0    0]
    [   0    0  1/2  1/2  1/2  1/2    0    0]
    [   0    0 -1/2  1/2  1/2 -1/2    0    0]
    [-1/2  1/2    0    0    0    0 -1/2  1/2]
    [   0    0 -1/2 -1/2  1/2  1/2    0    0]
    [-1/2 -1/2    0    0    0    0 -1/2 -1/2]
    [-1/2 -1/2    0    0    0    0  1/2  1/2]
    [ 1/2 -1/2    0    0    0    0 -1/2  1/2]

    [ 1/4  1/4  1/4 -1/4 -3/4 -1/4 -1/4 -1/4]
    [-1/4 -1/4  3/4  1/4 -1/4  1/4  1/4  1/4]
    [-1/4 -1/4 -1/4  1/4 -1/4  1/4  1/4 -3/4]
    [-1/4 -1/4 -1/4  1/4 -1/4  1/4 -3/4  1/4]
    [ 1/4 -3/4  1/4 -1/4  1/4 -1/4 -1/4 -1/4]
    [ 3/4 -1/4 -1/4  1/4 -1/4  1/4  1/4  1/4]
    [-1/4 -1/4 -1/4  1/4 -1/4 -3/4  1/4  1/4]
    [-1/4 -1/4 -1/4 -3/4 -1/4  1/4  1/4  1/4]
```

(continues on next page)

(continued from previous page)

```
    [1 0 0 0 0 0 0 0]
    [0 1 0 0 0 0 0 0]
    [0 0 1 0 0 0 0 0]
    [0 0 0 1 0 0 0 0]
    [0 0 0 0 1 0 0 0]
    [0 0 0 0 0 1 0 0]
    [0 0 0 0 0 0 0 1]
    [0 0 0 0 0 0 1 0]
> [ #Orbit(A, b) : b in Basis(L) ];
[ 2160, 240, 240, 240, 240, 240, 240, 240 ]
> AutomorphismGroup(L: NaturalAction) eq A;
true
```

Example: Automorphism Group of a 19-Dimensional Lattice (ex-c6d08f)

```
> L := Lattice("Lambda", 19);
> time G := AutomorphismGroup(L);
Time: 0.300
> #G;
23592960
> DS := DerivedSeries(G);
> [ #DS[i]/#DS[i+1] : i in [1..#DS-1] ];
[ 4, 3, 4 ]
> [ IsElementaryAbelian(DS[i]/DS[i+1]) : i in [1..#DS-1] ];
[ true, true, true ]
> H := DS[#DS];
> C := Core(G, Sylow(H, 2));
> Q := H/C; #Q, IsSimple(Q);
60 true
> LS := LowerCentralSeries(C);
> [ #LS[i]/#LS[i+1] : i in [1..#LS-1] ];
[ 256, 16, 2 ]
```

Hence, $G := \mathrm{Aut}(\Lambda_{19})$ has a series of normal subgroups with factors $2^2, 3, 2^2, A_5, 2^8, 2^4, 2$.

IsIsometric(L, M): Lat, Lat -> BoolElt, AlgMatElt

IsIsomorphic(L, M): Lat, Lat -> BoolElt, AlgMatElt

IsProperlyIsometric(L, M): Lat, Lat -> BoolElt, AlgMatElt

```
BacherDepth    : RngIntElt                    Default: 0
LeftGenerators : [ GrpMatElt ]                Default:
RightGenerators: [ GrpMatElt ]                Default:
LeftVectors    : Mtrx                         Default:
RightVectors   : Mtrx                         Default:
Decomposition  : BoolElt                      Default: false
NaturalAction  : BoolElt                      Default: false
```

This function determines whether the lattices L and M are (properly) isometric. The method is a backtrack search analogous to the one used to compute the automorphism group of a lattice. If the lattices are isometric, the function returns a transformation matrix T as a second return value such that $F_2 = TF_1T^{tr}$, where F_1 and F_2 are the Gram matrices of L and M, respectively. If they are properly isometric, return such T with $\det(T) = 1$.

For isometric lattices the cost of finding an isometry is roughly the cost of finding one automorphism of the lattice. Again, the computation may be sped up by using the additional invariants described for the automorphism group computation.

In many applications one will check whether a lattice is isometric to one for which the automorphism group is already known. In this situation the automorphism group of the second lattice can be made available by setting `RightGenerators := Q`, where Q is a set or sequence containing the generators of the group. Note however, that for isometric lattices this may slow down the computation, since generators for stabilizers have to be recomputed. Similarly, generators for an automorphism group of the first lattice may be supplied as `LeftGenerators`.

Corresponding to the `Vectors` parameter for automorphism group calculation, the parameters `LeftVectors` and `RightVectors` allow the user to. As above, left refers to the first lattice, right to the second. Either both of these parameters can be set, or neither, an error results if just one is set. The restrictions on what constitutes correct values for these parameters are as for the `Vectors` parameter above. For correct isometry testing, the vectors given must be such that any isometry will map the left vectors into the union of the right vectors and their negatives. These conditions are not checked in the code, they are the responsibility of the user.

Corresponding to the `NaturalAction` parameter for automorphism group calculation, the parameter `NaturalAction` allows the user to get the natural action of the isometry on the basis vectors (not the coordinate vectors); note that in this case the resulting matrix group will have (non-integral) rational entries in general, even though the image under the isometry of an integral basis vector will always be integral.

`IsIsometric(L, F1, M, F2): Lat, [ AlgMatElt ], Lat, [ AlgMatElt ] -> BoolElt, AlgMatElt`

`IsIsomorphic(L, F1, M, F2): Lat, [ AlgMatElt ], Lat, [ AlgMatElt ] -> BoolElt, AlgMatElt`

`IsIsometric(L, M): Lat, Lat -> BoolElt, AlgMatElt`

`IsIsomorphic(L, M): Lat, Lat -> BoolElt, AlgMatElt`

```
BacherDepth    : RngIntElt                    Default: 0
LeftGenerators : [ GrpMatElt ]                Default:
RightGenerators: [ GrpMatElt ]                Default:
LeftVectors    : Mtrx                         Default:
RightVectors   : Mtrx                         Default:
```

This function determines whether the lattices L and M are isometric with an isometry respecting also additional bilinear forms given by the sequences of Gram matrices F_1 and F_2. The return values and parameters are as above.

`IsIsometric(F1, F2): [ AlgMatElt ], [ AlgMatElt ] -> BoolElt, AlgMatElt`

`IsIsomorphic(F1, F2): [ AlgMatElt ], [ AlgMatElt ] -> BoolElt, AlgMatElt`

```
BacherDepth    : RngIntElt                    Default: 0
LeftGenerators : [ GrpMatElt ]                Default:
RightGenerators: [ GrpMatElt ]                Default:
LeftVectors    : Mtrx                         Default:
RightVectors   : Mtrx                         Default:
```

For two sequences of F_1 and F_2 of Gram matrices, determine whether a simultaneous isometry exists, i.e., a matrix T such that $TF_1[i]T^{tr} = F_2[i]$ for i in $[1..\#F_1]$. The first form in both sequences must be positive definite. The return values and parameters are as above.

Example: Isometry of Barnes-Wall Lattices `(ex-b3397d)`

We construct the 16-dimensional Barnes-Wall lattice in two different ways and show that the so-obtained lattices are isometric.

```
> L := Lattice("Lambda", 16);
> LL := Lattice(ReedMullerCode(1, 4), "B");
> time bool, T := IsIsometric(L, LL : Depth := 4);
Time: 2.029
```

(continues on next page)

(continued from previous page)

```
> bool;
true
> T * GramMatrix(L) * Transpose(T) eq GramMatrix(LL);
true
```

We can also show that L is a 2-modular lattice (i.e., isometric to its rescaled dual).

```
> IsIsometric(L, Dual(L));
true
[ 0  1  1 -1  1 -1  0  0 -1  1 -1  0  0  0  0  0]
[-2 -3 -4  1 -2  3 -1 -2  0  1 -1 -1  1  1  1 -1]
[-1 -1 -1  1  0 -1  0 -1  0  2 -1 -1  1  1  1 -1]
[ 0  1  1 -1  1  0  0  0 -1  0  0  0  0  0  0  0]
[ 0 -1 -2  0 -1  2 -1 -1  0  1 -1  0  1  0  0  0]
[ 1  2  2  0  2 -3  0  0 -2  4 -2 -1  1  1 -1 -1]
[-1 -1 -2  0  0  1  0 -1  0  0  0 -1  1  1  1 -1]
[ 1  2  3 -1  2 -3  1  1 -1  0  0  0  0  0 -1  1]
[ 0  1  1  0  2 -3  0 -1 -2  4 -2 -2  1  1  1 -1]
[ 0 -1 -2  0 -2  3 -1  0  1 -1  0  1  0  0 -1  0]
[ 0  0  1  1  0 -1  1  1  1 -1  0  1  0  0 -1  0]
[ 0 -1 -1  1 -2  2 -1  0  1  0  0  1  0 -1 -1  0]
[ 0  0  0  0  0  0  0  1  1 -1  0  1  0  0 -1  0]
[ 0 -1 -2  0 -2  3  0  0  1 -2  0  1  1  0 -2  1]
[ 0  1  1 -1  1 -1  0  0 -1  1  0 -1  0  0  1  0]
[ 0  0 -1 -1  0  1  0 -1 -1  1 -1 -1  1  1  0  0]
```

33.2.1 Automorphism Group and Isometry Testing over $\mathbb{F}_q[t]$

Let q be some power of an odd prime. A bilinear form b over $\mathbb{F}_q[t]$ is said to be definite if the corresponding quadratic form is anisotropic over the completion of $\mathbb{F}_q(t)$ at the infinite place $(1/t)$.

The functions in this section compute automorphism groups and isometries of definite bilinear forms over $\mathbb{F}_q[t]$.

`DominantDiagonalForm(X): Mtrx[RngUPol] -> Mtrx, Mtrx, GrpMat, FldFin`

```
Canonical     : Bool                    Default: false
ExtensionField: FldFin                  Default:
```

Let X be a symmetric $n \times n$-matrix of rank n over a polynomial ring $K[t]$ where K denotes a field of characteristic different from 2. The function returns a symmetric matrix G and some $T \in \mathrm{GL}(n, K[t])$ such that $G = TXT^{tr}$ has dominant diagonal. I.e. the degrees of the diagonal entries of G are ascending and the degree of a non-diagonal entry is less than the degrees of the corresponding diagonal entries (see [Ger03]).

If K is a finite field and X represents a definite form and `Canonical` is set to `true`, then the form G will be unique and the third return value will be the automorphism group of G i.e. the stabilizer of G in $\mathrm{GL}(n, K[t])$. The algorithm employed is [Kir12]. Note however, the uniqueness depends on some internal choices being made. Thus the fourth return value is a finite field E which must be given as the optional argument `ExtensionField` in subsequent runs over K to ensure that results are compatible (c.f. the following example). In particular, the defining polynomial and the primitive element of E are important for the uniqueness.

Example: Isometry Testing over a Polynomial Ring `(ex-898cdb)`

We test whether two definite forms over $\mathbb{F}_q[t]$ are isometric.

```
> R<t> := PolynomialRing( GF(5) );
> X1:= SymmetricMatrix([ t^3, t+1, 2*t^2+2*t+2 ] );
> X2:= SymmetricMatrix([ t^3, t^4+2*t+2, t^5+2*t^2+2*t+3 ]);
> G1, T1, Aut, E:= DominantDiagonalForm(X1 : Canonical);
> T1 * X1 * Transpose(T1) eq G1;
true
> GG:= [ Matrix(g) : g in Generators(Aut) ];
> forall{g : g in GG | g * G1 * Transpose(g) eq G1 };
true
```

So the form G_1 is invariant under Aut. Now we reduce the second form X_2. To be able to compare the results, we have to provide the field E from above.

```
> G2, T2 := DominantDiagonalForm(X2 : Canonical, ExtensionField:= E);
> G1 eq G2;
true
```

Thus the two forms X_1 and X_2 are isometric and $T_1^{-1}T_2$ is an isometry.

AutomorphismGroup(G): Mtrx[RngUPol] -> GrpMat, FldFin

ExtensionField: FldFin	Default:

Computes the automorphism group of the definite bilinear form given by the symmetric matrix G over $\mathbb{F}_q[t]$.

The second return value is a finite field as explained in DominantDiagonalForm above. It may be supplied for later calls over the same ground field $\mathbb{F}_q$ via the optional argument ExtensionField to safe some time if q is large. The correctness of the algorithm does not depend on it.

IsIsometric(G1, G2): Mtrx[RngUPol], Mtrx[RngUPol] -> BoolElt, Mtrx, FldFin

ExtensionField: FldFin	Default:

Tests whether two definite bilinear forms over $\mathbb{F}_q[t]$ are isometric. If so, the second return value is a matrix $T \in \mathrm{GL}(n, q)$ such that $TG1T^{tr} = G2$.

The third return value is a finite field as explained in DominantDiagonalForm above. It may be supplied for later calls over the same ground field $\mathbb{F}_q$ via the optional argument ExtensionField to safe some time if q is large. The correctness of the algorithm does not depend on it.

ShortestVectors(G): Mtrx[RngUPol] -> SeqEnum

Returns a sequence Q which contains the shortest non-zero vectors with respect to a given definite bilinear form G over $\mathbb{F}_q[t]$ where q is odd. The sequence Q contains tuples $< v, r >$ where v is a shortest vector and r denotes its norm with respect to G.

ShortVectors(G, B): Mtrx[RngUPol], RngIntElt -> SeqEnum

Let G be a definite bilinear form of rank n over $\mathbb{F}_q[t]$ for some odd q. The function returns a sequence Q which contains all vectors in $\mathbb{F}_q[t]^n$ whose norm with respect to G is at most B. The sequence Q contains tuples $< v, r >$ where v is such a short vector and r denotes its norm with respect to G.

33.3 Lattices from Matrix Groups

In Magma a G-lattice L is a lattice upon which a finite integral matrix group G acts by right multiplication. Each G-lattice L has references to both the original ("natural") group G which acts on the standard lattice in which L is embedded and also the reduced group of L which is the reduced representation of G on the basis of L.

33.3.1 Creation of G-Lattices

The following functions create G-lattices. Note that the group G must be a finite integral matrix group.

`Lattice(G): GrpMat -> Lat`

Given a finite integral matrix group G, return the standard G-lattice (with standard basis and rank equal to the degree of G).

`LatticeWithBasis(G, B): GrpMat, ModMatRngElt -> Lat`

Given a finite integral matrix group G and a non-singular matrix B whose row space is invariant under G (i.e., $Bg = T_g B$ for each $g \in G$ where T_g is a unimodular integral matrix depending on g), return the G-lattice with basis matrix B. (The number of columns of B must equal the degree of G; G acts naturally on the lattice spanned by B.)

`LatticeWithBasis(G, B, M): GrpMat, ModMatRngElt, AlgMatElt -> Lat`

Given a finite integral matrix group G, a non-singular matrix B whose row space is invariant under G (i.e., $Bg = T_g B$ for all $g \in G$ where T_g is a unimodular integral matrix depending on g) and a positive definite matrix M invariant under G (i.e., $gMg^{tr} = M$ for all $g \in G$) return the G-lattice with basis matrix B and inner product matrix M. (The number of columns of B must equal the degree of G and both the number of rows and the number of columns of M must equal the degree of G; G acts naturally on the lattice spanned by B and fixes the Gram matrix of the lattice).

`LatticeWithGram(G, F): GrpMat, AlgMatElt -> Lat`

Given a finite integral matrix group G and a positive definite matrix F invariant under G (i.e., $gFg^{tr} = F$ for all $g \in G$) return the G-lattice with standard basis and inner product matrix F (and thus Gram matrix F). (Both the number of rows and the number of columns of M must equal the degree of G; G fixes the Gram matrix of the returned lattice).

33.3.2 Operations on G-Lattices

The following functions provide basic operations on G-lattices.

`IsGLattice(L): Lat -> GrpMat`

Given a lattice L, return whether L is a G-lattice (i.e., there is a group associated with L).

`Group(L): Lat -> GrpMat`

Given a G-lattice L, return the matrix group of the (reduced) action of G on L. The resulting group thus acts on the coordinate lattice of L (like the automorphism group).

`NumberOfActionGenerators(L): Lat -> RngIntElt`

`Nagens(L): Lat -> RngIntElt`

Given a G-lattice L, return the number of generators of G.

`ActionGenerator(L, i): Lat, RngIntElt -> GrpMat`

Given a G-lattice L, return the i-th generator of the (reduced) action of G on L. This is the reduced action of the i-th generator of the original group G (which may be the identity matrix).

`NaturalGroup(L): Lat -> GrpMat`

Given a G-lattice L, return the matrix group of the (natural) action of G on L. The resulting group thus acts on L naturally.

`NaturalActionGenerator(L, i): Lat, RngIntElt -> GrpMat`

Given a G-lattice L, return the i-th generator of the natural action of G on L. This is simply the i-th generator of the original group G.

33.3.3 Invariant Forms

The functions in this section compute invariant forms for G-lattices.

`InvariantForms(L): Lat -> [ AlgMatElt ]`

For a G-lattice L, return a basis for the space of invariant bilinear forms for G (represented by their Gram matrices) as a sequence of matrices. The first entry of the sequence is a positive definite symmetric form for G.

`InvariantForms(L, n): Lat, RngIntElt -> [ AlgMatElt ]`

For a G-lattice L, return a sequence consisting of $n \geq 0$ invariant bilinear forms for G.

`SymmetricForms(L): Lat -> [ AlgMatElt ]`

For a G-lattice L, return a basis for the space of symmetric invariant bilinear forms for G. The first entry of the sequence is a positive definite symmetric form of G.

`SymmetricForms(L, n): Lat, RngIntElt -> [ AlgMatElt ]`

For a G-lattice L, return a sequence of $n \geq 0$ independent symmetric invariant bilinear forms for G. The first entry of the first sequence (if $n > 0$) is a positive definite symmetric form for G.

`AntisymmetricForms(L): Lat -> [ AlgMatElt ]`

For a G-lattice L, return a basis for the space of antisymmetric invariant bilinear forms for G.

`AntisymmetricForms(L, n): Lat, RngIntElt -> [ AlgMatElt ]`

For a G-lattice L, return a sequence of $n \geq 0$ independent antisymmetric invariant bilinear forms for G.

`NumberOfInvariantForms(L): Lat -> RngIntElt, RngIntElt`

For a G-lattice L, return the dimension of the space of (symmetric and anti-symmetric) invariant bilinear forms for G. The algorithm uses a modular method which is always correct and is faster than the actual computation of the forms.

`NumberOfSymmetricForms(L): Lat -> RngIntElt`

For a G-lattice L, return the dimension of the space of symmetric invariant bilinear forms for G.

`NumberOfAntisymmetricForms(L): Lat -> RngIntElt`

For a G-lattice L, return the dimension of the space of antisymmetric invariant bilinear forms for G.

`PositiveDefiniteForm(L): Lat -> AlgMatElt`

For a G-lattice L, return a positive definite symmetric form for G. This is a positive definite matrix F such that $gFg^{tr} = F$ for all $g \in G$.

33.3.4 Endomorphisms

The functions in this subsection compute endomorphisms of G-lattices. This is done by approximating the averaging operator over the group and applying it to random elements.

`EndomorphismRing(L): Lat -> AlgMat`

For a G-lattice L, return the endomorphism ring of L as a matrix algebra over $\mathbb{Q}$.

`Endomorphisms(L, n): Lat, RngIntElt -> [ AlgMatElt ]`

For a G-lattice L, return a sequence containing n independent endomorphisms of L as elements of the corresponding matrix algebra over $\mathbb{Q}$. n must be in the range $[0..d]$, where d is the dimension of the endomorphism ring of L. This function may be useful in situations where the full endomorphism algebra is not required, e.g., to split a reducible lattice.

`DimensionOfEndomorphismRing(L): Lat -> RngIntElt`

Return the dimension of the endomorphism algebra of the G-lattice L by a modular method (which always yields a correct answer).

`CentreOfEndomorphismRing(L): Lat -> AlgMat`

For a G-lattice L, return the centre of the endomorphism ring of L as a matrix algebra over $\mathbb{Q}$.

This function can be used to split a reducible lattice into its homogeneous components.

`CentralEndomorphisms(L, n): Lat, RngIntElt -> [ AlgMatElt ]`

For a G-lattice L, return a sequence containing n independent central endomorphisms of L as elements of the corresponding matrix algebra over $\mathbb{Q}$. n must be in the range $[0..d]$, where d is the dimension of the centre of the endomorphism ring of L.

`DimensionOfCentreOfEndomorphismRing(L): Lat -> RngIntElt`

Return the dimension of the centre of the endomorphism algebra of the G-lattice L by a modular method (which always yields a correct answer).

33.3.5 G-invariant Sublattices

The functions in this section compute G-invariant sublattices of a given G-lattice L.

For a fixed prime p, the algorithm constructs the maximal G-invariant sublattices of L as kernels of $\mathbb{F}_pG$-epimorphisms $L/pL \to S$ for some simple $\mathbb{F}_pG$-module S as described in [Ple74].

Iterating this process yields all G-invariant sublattices of L whose index in L is a p-power. Finally, intersecting lattices of coprime index yields all sublattices of L.

`Sublattices(G, Q): GrpMat, [ RngIntElt ] -> [ Lat ], BoolElt`

`Sublattices(G, Q): [ Mtrx ], [ RngIntElt ] -> [ Lat ], BoolElt`

`Sublattices(L, Q): Lat, [ RngIntElt ] -> [ Lat ], BoolElt`

```
Limit      : RngIntElt                  Default: Infinity()
Levels     : RngIntElt                  Default: Infinity()
Projections: [Mtrx]                     Default: []
```

Given either

(a) an integral matrix group G with natural lattice $L = \mathbb{Z}^n$

(b) a sequence G of integral matrices generating a $\mathbb{Z}$-order in $\mathbb{Q}^{n\times n}$ with natural lattice $L = \mathbb{Z}^n$

(c) a G-lattice L in $\mathbb{Q}^n$.

together with a set or sequence Q of primes, compute the G-invariant sublattices of L (as a sequence) which are not contained in pL for any $p \in Q$ and whose index in L is a product of elements of Q.

This set of G-invariant sublattices of L is finite if and only if $\mathbb{Q}_p \otimes L$ is irreducible as a $\mathbb{Q}_pG$-module for all $p \in Q$.

Setting the parameter `Limit := n` will terminate the computation after n sublattices have been found.

Setting the parameter `Levels := n` will only compute sublattices M such that L/M has at most n composition factors.

The optional parameter `Projections` can be a sequence of n by n matrices that describe projections on $\mathbb{Q}^n$ that map L to itself. In this case, Magma will only compute those sublattices of L which have the same images under the projections as L does.

The second return value indicates whether the returned sequence contains all such sublattices or not.

`Sublattices(G, p): GrpMat, RngIntElt -> [ Lat ], BoolElt`

`Sublattices(L, p): Lat, RngIntElt -> [ Lat ], BoolElt`

```
Limit      : RngIntElt                  Default: Infinity()
Levels     : RngIntElt                  Default: Infinity()
Projections: [Mtrx]                     Default: []
```

The same as the above where the set Q consists only of the given ptime p.

`Sublattices(G): GrpMat -> [ Lat ], BoolElt`

Sublattices(L): Lat -> [Lat], BoolElt

```
Limit      : RngIntElt                    Default: Infinity()
Levels     : RngIntElt                    Default: Infinity()
Projections: [Mtrx]                       Default: []
```

For an integral matrix group G or a G-lattice L this intrinsic equals the one above with Q taken to be the prime divisors of the order of G.

SublatticeClasses(G): GrpMat -> [Lat]

```
MaximalOrders: BoolElt                    Default: false
```

For an integral matrix group G returns representatives for the isomorphism classes of G-invariant lattices (i.e. the orbits under the unit group of the endomorphism ring E of G).

If `MaximalOrders` is set to `true`, only sublattice classes which are invariant under some maximal order of E are considered.

Currently the function requires E to be a field.

Example: Sublattices of a G-Lattice (ex-ecd505)

We construct sublattices of the standard G-lattice where G is an absolutely irreducible degree-8 integral matrix representation of the group $\mathrm{GL}(2,3) \times \mathrm{S}_3$.

We first define the group G.

```
> G := MatrixGroup<8, IntegerRing() |
>    [-1,  0,  0,  0,  0,  0,  0,  0,
>      0,  0, -1,  0,  0,  0,  0,  0,
>      0,  0,  0,  1,  0,  0,  0,  0,
>      0,  1,  0,  0,  0,  0,  0,  0,
>     -1,  0,  0,  0,  1,  0,  0,  0,
>      0,  0, -1,  0,  0,  0,  1,  0,
>      0,  0,  0,  1,  0,  0,  0, -1,
>      0,  1,  0,  0,  0, -1,  0,  0],
>
>    [ 0,  0,  0,  0,  0,  0,  0,  1,
>      0,  0,  0,  0,  0,  0,  1,  0,
>      0,  0,  0,  0, -1,  0,  0,  0,
>      0,  0,  0,  0,  0,  1,  0,  0,
>      0,  0,  0, -1,  0,  0,  0,  1,
>      0,  0, -1,  0,  0,  0,  1,  0,
>      1,  0,  0,  0, -1,  0,  0,  0,
>      0, -1,  0,  0,  0,  1,  0,  0]>;
```

We next compute the unique positive definite form F fixed by G.

```
> time F := PositiveDefiniteForm(G);
Time: 0.050
> F;
[2 0 0 0 1 0 0 0]
[0 2 0 0 0 1 0 0]
[0 0 2 0 0 0 1 0]
[0 0 0 2 0 0 0 1]
[1 0 0 0 2 0 0 0]
[0 1 0 0 0 2 0 0]
[0 0 1 0 0 0 2 0]
[0 0 0 1 0 0 0 2]
```

We now compute all sublattices of the standard G-lattice.

```
> time Sub := Sublattices(G);
Time: 0.370
> #Sub;
18
```

For each sublattice we compute the invariant positive definite form for the group given by the action of G on the sublattice.

```
> PrimitiveMatrix := func<X |
>     P ! ((ChangeRing(P, RationalField()) ! X) / GCD(Eltseq(X)))
>         where P is Parent(X)>;
> FF := [PrimitiveMatrix(B * F * Transpose(B))
>             where B is BasisMatrix(L): L in Sub];
```

We next create the sequence of all the lattices whose Gram matrices are given by the (LLL-reduced) forms.

```
> Sub := [LatticeWithGram(LLLGram(F)) : F in FF];
> #Sub;
18
```

We now compute representatives for the $\mathbb{Z}$-isomorphism classes of the sequence of lattices.

```
> Rep := [];
> for L in Sub do
>     if forall{LL: LL in Rep | not IsIsometric(L, LL)} then
>         Append(~Rep, L);
>     end if;
> end for;
> #Rep;
4
```

Thus there are 4 non-isomorphic sublattices. We note the size of the automorphism group, the determinant, the minimum, and the kissing number of each lattice. (In fact, the automorphism groups of these 4 lattices happen to be maximal finite subgroups of $\mathrm{GL}(8,\mathbb{Q})$ and all have $\mathrm{GL}(2,3)\times \mathrm{S}_3$ as a common irreducible subgroup.)

```
> time A := [AutomorphismGroup(L) : L in Rep];
Time: 0.240
> [#G: G in A];
[ 497664, 6912, 696729600, 2654208 ]
> [Determinant(L): L in Rep];
[ 81, 1296, 1, 16 ]
> [Minimum(L): L in Rep];
[ 2, 4, 2, 2 ]
> [KissingNumber(L): L in Rep];
[ 24, 72, 240, 48 ]
```

Finally, we note that each lattice is isomorphic to a standard construction based on root lattices.

```
> l := IsIsometric(Rep[1],
>        TensorProduct(Lattice("A", 2), StandardLattice(4))); l;
true
> l := IsIsometric(Rep[2],
>        TensorProduct(Lattice("A", 2), Lattice("F", 4))); l;
true
> l := IsIsometric(Rep[3], Lattice("E", 8)); l;
```

(continues on next page)

(continued from previous page)

```
true
> l := IsIsometric(Rep[4],
>        TensorProduct(Lattice("F", 4), StandardLattice(2))); l;
```

Example: Sublattices with Projections (ex-e8ccf0)

This example illustrates the optional argument `Projections`.

```
> G := MatrixGroup<4, IntegerRing() |
> [ -1, 0, 1, 0, 0, -1, 1, -3, -1, 0, 0, 0, 0, 0, 0, 1 ],
> [ -1, 0, 0, 0, -3, 2, 0, 3, 0, 0, -1, 0, 1, -1, 0, -1 ] >;
> E := EndomorphismRing(G);
> I := CentralIdempotents(ChangeRing(E, RationalField())); I;
[
    [ 0  0  0  0]
    [-1  1  0  0]
    [ 0  0  0  0]
    [ 0  0  0  1],

    [1 0 0 0]
    [1 0 0 0]
    [0 0 1 0]
    [0 0 0 0]
]
```

Since the central idempotents are all integral, they map the standard lattice $\mathbb{Z}^n$ to itself. Even though this group G fixes infinitely many sublattices of Z^n (even up to scalar multiples), there can only be finitely many which have the same images under the central idempotents as $\mathbb{Z}^n$.

```
> S := Sublattices(G : Projections:= I); #S;
3
```

So in this case there are only three such lattices. To check that the lattices do project correctly, we can use

```
> I := [ Matrix(Integers(), i) : i in I ];
> Images := [ [Image(BasisMatrix(s) * i) : i in I] : s in S ];
> #Set(Images) eq 1;
true
```

33.3.6 Lattice of Sublattices

Magma can construct the lattice V of all G-invariant sublattices of the standard lattice $L = \mathbb{Z}^n$. Various properties of the lattice V may then be examined. Magma only stores the primitive sublattices of L, i.e. those sublattices that are not contained in kL for some $k > 1$.

In general, G fixes infinitely many primitive lattices. Thus one has to limit the number of sublattices to be constructed just as in the `Sublattice` intrinsic. In this case, all operations on V like coercions, intersections, sums etc. assume that the result of the operation is again is a scalar multiple of some element stored in V.

The lattice V has type `LatLat` and elements of V have type `LatLatElt` and are numbered from 1 to n where n is the number of primitive sublattices of L that have been constructed in the beginning.

Creating the Lattice of Sublattices

`SublatticeLattice(G, Q): GrpMat, [ RngIntElt ] -> LatLat, BoolElt`

`SublatticeLattice(G, Q): [ Mtrx ], [ RngIntElt ] -> LatLat, BoolElt`

```
Limit      : RngIntElt                    Default: Infinity()
Levels     : RngIntElt                    Default: Infinity()
Projections: [ Mtrx ]                     Default: []
```

Given either an integral matrix group G of degree n or a sequence G of integral matrices generating a $\mathbb{Z}$-order in $\mathbb{Q}^{n\times n}$ together with a set or sequence Q of primes, compute the G-invariant sublattices of $\mathbb{Z}^n$ (as a sequence) which are not contained in $p\mathbb{Z}^n$ for any $p \in Q$ and whose index in $\mathbb{Z}^n$ is a product of elements of Q.

The second return value indicates whether all G-invariant lattices have been constructed.

The optional parameters are the same as for the `Sublattices` intrinsic.

`SublatticeLattice(G, p): GrpMat, RngIntElt -> LatLat, BoolElt`

`SublatticeLattice(G, p): [ Mtrx ], RngIntElt -> LatLat, BoolElt`

```
Limit      : RngIntElt                    Default: Infinity()
Levels     : RngIntElt                    Default: Infinity()
Projections: [ Mtrx ]                     Default: []
```

Same as above where the set Q consists only of the given prime p.

`SublatticeLattice(G): GrpMat -> LatLat, BoolElt`

```
Limit      : RngIntElt                    Default: Infinity()
Levels     : RngIntElt                    Default: Infinity()
Projections: [ Mtrx ]                     Default: []
```

Same as above where the set Q is taken to be set of prime divisors of the order of the group G.

Example: Creating a Lattice of Sublattices (ex-bee7b0)

This example shows how to create a lattice of sublattices.

```
> G:= sub< GL(2, Integers()) | [0,1,-1,0] >;
> V:= SublatticeLattice(G); V;
Lattice of 2 sublattices
```

Operations on the Lattice of Sublattices

In the following, V is a lattice of G-invariant lattices for some group or $\mathbb{Z}$-order G and Q denotes the set of primes that where used to create V.

`# V: LatLat -> RngIntElt`

The number of (primitive) lattices stored in V.

`V ! i: LatLat, RngIntElt -> LatLatElt`

The i-th element of the lattice V with respect to the internal labeling.

`V ! M: LatLat, Lat -> LatLatElt`

`V ! M: LatLat, Mtrx -> LatLatElt`

Given a (basis matrix of some) G-invariant lattice M, create the element of the lattice V corresponding to M.

`NumberOfLevels(V): LatLat -> RngIntElt`

The number of different levels (layers) stored in V. Note that levels are counted starting from 0.

`Level(V, i): LatLat, RngIntElt -> [ LatLatElt ]`

The primitive lattices stored at the i-th level (layer). Note that levels are counted starting from 0.

`Levels(v): LatLat -> [ [LatLatElt] ]`

The i-th entry of the result is a sequence of the primitive lattice elements lying on the $i-1$-th level.

`Primes(V): LatLat -> [ RngIntElt ]`

The primes that where used to create V.

`Constituents(V): LatLat -> SeqEnum`

A sequence containing the constituents (simple $\mathbb{F}_p G$ - modules) that where used during the construction of the G-lattices in V.

`IntegerRing() ! e: RngInt, LatLatElt -> RngIntElt`

The integer corresponding to lattice element e.

`e + f: LatLatElt, LatLatElt -> LatLatElt`

The sum of the lattice elements e and f.

`e meet f: LatLatElt, LatLatElt -> LatLatElt`

The intersection of the lattice elements e and f.

`e eq f: LatLatElt, LatLatElt -> BoolElt`

Tests whether e and f are equal.

`MaximalSublattices(e): LatLatElt -> [ LatLatElt ], [ RngIntElt ]`

The sequence S of maximal sublattices of e having index p for some $p \in Q$. The second return value is a list C of integers such that $S[i]/e$ is isomorphic to the $C[i]$-th constituent of V. The ordering of the constituents is the same as in the `Constituents` intrinsic.

`MinimalSuperlattices(e): LatLatElt -> [ LatLatElt ], [ RngIntElt ]`

The sequence S of minimal superlattices of e in which e has index p for some $p \in Q$. The second return value is a list C of integers such that $e/S[i]$ is isomorphic to the $C[i]$-th constituent of V. The ordering of the constituents is the same as in the `Constituents` intrinsic.

`Lattice(e): SubModLatElt -> Lat`

The G-lattice corresponding to e.

`BasisMatrix(e): SubModLatElt -> Mtrx`

`Morphism(e): SubModLatElt -> Mtrx`

The basis matrix of the G-lattice corresponding to e.

Example: Invariant Lattices of the Root Lattice A5 (ex-f3a5a3)

Let G be the automorphism group of the root lattice A_5. Since G is absolutely irreducible, it fixes only finitely many lattices up to scalars. We explore them.

```
> G:= AutomorphismGroup(Lattice("A", 5));
> FactoredOrder(G);
[ <2, 5>, <3, 2>, <5, 1> ]
> #SublatticeLattice(G, 5);
1
```

Hence there are no primitive sublattices between L and $5L$. Hence it suffices to check only the lattices at 2 and 3 the two remaining prime divisors of the order of G.

```
> V:= SublatticeLattice(G, {2,3}); #V;
4
> M:= MaximalSublattices(V ! 1); M;
[
```

(continues on next page)

(continued from previous page)

```
    sublattice number 2,
    sublattice number 3
]
> V ! 2 meet V ! 3;
sublattice number 4
```

Moreover, the second and third lattice are (up to rescaling) dual to each other with respect to some G-invariant form.

```
> F:= PositiveDefiniteForm(G);
> L:= Dual(Lattice(BasisMatrix(V ! 2), F) : Rescale:= false);
> V ! L;
sublattice number 3 times 1/6
```

In particular, every G-invariant lattice can be constructed from lattice number 2 by taking scalar multiples, duals, sums, and intersections. For example the standard lattice can be written as:

```
> (V ! 2) + (V ! (6*L));
sublattice number 1
```

Example: Lattices Invariant under SL(2,7) `(ex-573832)`

Let G be the 8-dimensional (faithful) rational representation of $\mathrm{SL}(2,7)$. Its endomorphism ring E is isomorphic to $\mathbb{Q}(\sqrt{-7})$. We find all G-invariant lattices of G that are invariant under the maximal order M of E up to multiplication with elements in E. After this is done, we quickly obtain all finite subgroups of $\mathrm{GL}(8,\mathbb{Q})$ (up to conjugacy) that include a normal subgroup conjugate to G.

To shorten the example, we choose G such that the standard lattice L is already invariant under M.

```
> SetSeed(1);
> G:= MatrixGroup<8, IntegerRing() |
>    [ 0, 1, 0, 0, 0, 0, -1, 0, -1, 0, 0, 0, 0, 0, -1, 1,
>      0, 0, 0, 1, 0, 0, -1, 0, 0, 0, -1, 0, 0, 0, -1, 1,
>      0, 0, 0, 0, 0, 1, -1, 0, 0, 0, 0, 0, -1, 0, -1, 1,
>      0, 0, 0, 0, 0, 0, -1, 1, 0, 0, 0, 0, 0, 0, -2, 1 ],
>    [ 0, -1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0,
>     -1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, -1, 1,
>      0, 0, 0, -1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0,
>      0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1 ] >; #G;
336
> E:= EndomorphismRing(G);
> M:= MaximalOrder(ChangeRing(E, RationalField()));
> ok, M:= CanChangeUniverse(Basis(M), MatrixRing(Integers(), 8)); ok;
true
```

So L is M-invariant. The lattices at the primes 3 and 7 are multiples of L as we can see as follows:

```
> w7:= IntegralMatrix(E.2 - Trace(E.2)/8);
> w7 div:= GCD( Eltseq(w7) );                    // a square root of 7
> V:= SublatticeLattice([ Matrix(G.i) : i in [1..Ngens(G)] ] cat M, [3,7]); #V;
2
> V ! w7;
sublattice number 2
```

So it remains to check the lattices at 2. The two prime ideals in M over 2 are generated by `p` and `q` where

```
> p:= 1 - (w7+1) div 2;
> q:= (w7+1) div 2;
> Gens:= [ Matrix(G.i) : i in [1..Ngens(G)] ];
> V:= SublatticeLattice(Gens cat M, 2: Levels:= 3);
> Levels(V);
[
    [
        sublattice number 1
    ],
    [
        sublattice number 2,
        sublattice number 3
    ],
    [
        sublattice number 4,
        sublattice number 5,
        sublattice number 6
    ],
    [
        sublattice number 7,
        sublattice number 8,
        sublattice number 9,
        sublattice number 10
    ]
]
> [ V | BasisMatrix(V ! i)*x : i in [1..3], x in [p,q] ];
[
    sublattice number 4,
    sublattice number 7,
    sublattice number 8,
    sublattice number 6,
    sublattice number 9,
    sublattice number 10
]
```

So the lattice numbers $1, 2, 3$ and 5 represent the orbits of the action of E on the set of all MG-invariant lattices. Moreover, every matrix group N normalizing G acts on the MG-invariant lattices and (up to conjugacy) thus fixes one of these four lattices. If it fixes L, it also fixes $V!2 + V!3 = V!5$ and vice versa. Similarly, it fixes $V!2$ if and only if it fixes $V!3$.

```
> F:= PositiveDefiniteForm(G);
> N1:= Normalizer(AutomorphismGroup(LatticeWithGram(F)), G); #N1;
672
> A:= AutomorphismGroup(Lattice(BasisMatrix(V ! 2), F) : NaturalAction);
> N2:= Normalizer(A, ChangeRing(G, Rationals())); #N2;
336
```

So `N1` (which is isomorphic to $2.L(2,7) : 2$) is up to conjugacy the only proper finite extension of G in $\mathrm{GL}(8, \mathbb{Q})$.

33.4 Bibliography

- [Ger03] Larry J. Gerstein. Definite quadratic forms over $\mathbb{F}_q[X]$. *J. Algebra*, 268(1):252–263, 2003.
- [Kir12] M. Kirschmer. A normal form for definite quadratic forms over $\mathbb{F}_q[t]$. *Math. Comp.*, 81:1619–1634, 2012.
- [Ple74] Wilhelm Plesken. *Beiträge zur Bestimmung der endlichen irreduziblen Untergruppen von*

GL(n,Z) und ihrer ganzzahligen Darstellungen. PhD thesis, RWTH Aachen, 1974.

- [PS97a] Wilhelm Plesken and Bernd Souvignier. Computing Isometries of Lattices. *J. Symbolic Comp.*, 24(3):327–334, 1997.

Chapter 34

Quadratic Forms

34.1 Introduction

This chapter describes miscellaneous functionality for fairly general quadratic forms. The main feature currently here is an implementation of Simon's algorithm for finding isotropic subspaces of integral forms.

In Magma, quadratic forms are generally represented either as multivariate polynomials, or as symmetric matrices.

34.2 Constructions and Conversions

`SymmetricMatrix(f): RngMPolElt -> Mtrx`

Given a multivariate polynomial f that is homogeneous of degree 2, this returns a symmetric matrix representing the same quadratic form.

`GramMatrix(L): Lat -> Mtrx`

The symmetric matrix giving the quadratic form on the lattice L.

`QuadraticForm(L): Lat -> RngMPolElt`

The quadratic form associated to the lattice L, as a multivariate polynomial.

`QuadraticForm(M): Mtrx -> RngMPolElt`

The quadratic form for a symmetric matrix M, as a multivariate polynomial.

```
QuadraticFormWithInvariants(n, d, F, N): RngIntElt, RngElt, { RngIntElt }, RngIntElt
                            -> AlgMatElt
QuadraticFormWithInvariants(n, d, F, N): RngIntElt, FldAlgElt, { RngOrdIdl }, [
                            RngIntElt ] -> AlgMatElt
```

Quadratic form of dimension n and determinant d that has Hasse invariants -1 at the primes in F. The number of negative entries of the i-th real signature is given by N_i.

34.3 Local Invariants

These commands calculate the standard invariants that characterize a quadratic form over the rationals. Definitions of the invariants may be found in Conway-Sloane [JC98], Chapter 15, Section 5.1.

`pSignature(f, p): RngMPolElt, RngIntElt -> RngIntElt`

`pSignature(M, p): Mtrx, RngIntElt -> RngIntElt`

`pSignature(L, p): Lat, RngIntElt -> RngIntElt`

The p-signature of the specified quadratic form over the rationals, where p is a prime number or -1 (designating the real place).

For odd primes p, this is defined by diagonalizing the form, and adding p-parts of these entries to 4 times the number of anti-squares (mod p) amongst these entries. The term "anti-square" modulo p denotes something that has: odd valuation at p; and the prime-to-p part, called u, has Kronecker symbol $(\frac{u}{p}) = -1$.

At $p = 2$ it is the sum of the odd parts of the diagonalized entries plus 4 times the number of anti-squares. In either case, the final answer is really only defined modulo 8 (this is so that p-signatures are invariant under rational equivalence).

At the real place, it is the difference between the number of positive and negative eigenvalues (the terminology here can be murky).

`Oddity(f): RngMPolElt -> RngIntElt`

`Oddity(L): Lat -> RngIntElt`

`Oddity(M): Mtrx -> RngIntElt`

This returns the 2-signature of the given quadratic form over the rationals.

`pExcess(f, p): RngMPolElt, RngIntElt -> RngIntElt`

`pExcess(M, p): Mtrx, RngIntElt -> RngIntElt`

`pExcess(L, p): Lat, RngIntElt -> RngIntElt`

The p-excess of the specified quadratic form over the rationals, where p is a prime number or -1 (designating the real place). The p-excess is the difference between the p-signature and dimension for odd primes (including -1), and is the negation of this for $p = 2$. The sum of p-excesses over all primes should be 0 modulo 8.

`WittInvariant(f, p): RngMPolElt, RngIntElt -> RngIntElt`

`WittInvariant(M, p): AlgMatElt, RngIntElt -> RngIntElt`

Calculates the Witt invariant over $\mathbb{Q}_p$ of the given quadratic form. Again the form must be defined over either the rationals or the integers. The result is returned as something in the set $\{-1, +1\}$. This is the class in the Brauer group of the Clifford algebra for even dimension, of the even Clifford algebra for odd dimension, see §3 of Chapter V of Lam [Lam05].

`WittInvariant(L, p): Lat, RngIntElt -> RngIntElt`

`WittInvariant(L, P): Lat, RngInt -> RngIntElt`

`WittInvariant(L, P): LatNF, RngOrdIdl -> RngIntElt`

`AmbientSpace: BoolElt` Default: `false`

The Witt invariant of the quadratic form at p (or P). If `AmbientSpace` is set to `true`, returns the Witt invariant of the ambient quadratic space.

`HasseInvariant(f, p): RngMPolElt, RngIntElt -> RngIntElt`

`HasseInvariant(M, p): AlgMatElt, RngIntElt -> RngIntElt`

`HasseMinkowskiInvariant(f, p): RngMPolElt, RngIntElt -> RngIntElt`

`HasseMinkowskiInvariant(M, p): AlgMatElt, RngIntElt -> RngIntElt`

Calculates the Hasse (or Hasse-Minkowski) invariant over $\mathbb{Q}_p$ of the given quadratic form. Again the form must be defined over either the rationals or the integers. The result is returned as something in the set $\{-1, +1\}$. One definition of this invariant is to diagonalize the form and then take the product (in our multiplicative notation) of the Hilbert symbols of the $\binom{n}{2}$ pairs of distinct nonzero diagonal entries, as in §5.3 of Chapter 15 of Conway-Sloane [JC98], which is what is implemented here. Another method would be to use a comparison of p-excesses with the standard form (also in Conway-Sloane). Starting with a p-adic input could lead to precision problems at the diagonalization step, and so is not allowed. Can also be called via `HasseMinkowskiInvariant`.

`HasseInvariant(L, p): Lat, RngIntElt -> RngIntElt`

`HasseMinkowskiInvariant(L, p): Lat, RngIntElt -> RngIntElt`

`HasseInvariant(L, P): Lat, RngInt -> RngIntElt`

`HasseInvariant(L, P): LatNF, RngOrdIdl -> RngIntElt`

HasseMinkowskiInvariant(L, P): Lat, RngInt -> RngIntElt

AmbientSpace: BoolElt		Default: false

The Hasse invariant of the quadratic form at p (or at P). If AmbientSpace is set to true, returns the Hasse invariant of the ambient quadratic space.

WittInvariants(f): RngMPolElt -> SeqEnum

WittInvariants(M): AlgMatElt -> SeqEnum

WittInvariants(L): Lat -> SeqEnum

HasseInvariants(f): RngMPolElt -> SeqEnum

HasseInvariants(M): AlgMatElt -> SeqEnum

HasseInvariants(L): Lat -> SeqEnum

HasseMinkowskiInvariants(f): RngMPolElt -> SeqEnum

HasseMinkowskiInvariants(M): AlgMatElt -> SeqEnum

HasseMinkowskiInvariants(L): Lat -> SeqEnum

Minimize: BoolElt		Default: false
AA : BoolElt		Default: false

Compute WittInvariant(f,p) or HasseInvariant(f,p), repsectively, for all bad primes p, and return the result of a sequence of tuples, each entry given by $\langle p, W_p(f)\rangle$. The set of bad primes includes the real place, the prime $p = 2$, and all primes that divide either the numerator or the denominator of the determinant or symmetric matrix associated to f. If Minimize is set, the primes greater than 2 must have odd valuation in the determinant to appear. If AA is set, the result is return as an associative array. Can also be called via HasseMinkowskiInvariants.

QuadraticFormInvariants(f): RngMPolElt -> FldElt, SetEnum, SeqEnum[RngIntElt]

QuadraticFormInvariants(M): AlgMatElt -> FldElt, SetEnum, SeqEnum[RngIntElt]

QuadraticFormInvariants(L): LatNF -> FldElt, SetEnum, SeqEnum[RngIntElt]

Minimize: BoolElt		Default: false
AA : BoolElt		Default: false

The determinant of the diagonalization, the set of Hasse-Minkowski invariants, and the number of negative eigenvalues (in each real place) of the quadratic form M. If AA is set, the Hasse-Minkowski invariants are returned as an associative array. If Minimize is set, the set of Hasse-Minkowski invariants is minimized, returning only the prime ideals for which the Hasse-Minkowski invariant is -1.

34.4 Isotropic Subspaces

IsotropicSubspace(f): RngMPolElt -> ModTupRng

IsotropicSubspace(M): Mtrx -> ModTupRng

This returns an isotropic subspace for the given quadratic form (which must be either integral or rational), which may be given either as a multivariate polynomial f or as a symmetric matrix M. The subspace returned is in many cases guaranteed to be a maximal totally isotropic subspace. More precisely (assuming that the form is nonsingular), upon writing (r, s) for the signature of f with $r \geq s$, the dimension of the space returned is at least $\min(r, s + 2) - 2$, which is maximum possible when $s \leq r + 2$. Since version 2.18, an improvement to the original algorithm has been appended, which typically (subject to a solvability criterion for a 4-dimensional subspace) enlarges the dimension of space by 1 when $r + s$ is even and $s \leq r + 2$.

The algorithm used is due to Simon (see [Sim05a]), and uses on the Bosma-Stevenhagen algorithm for the 2-part of the class groups of a quadratic field (see [BS96]). There is no corresponding intrinsic for a lattice: since the associated form is definite, there are no isotropic vectors.

Example: Isotropy and Witt Index of a Form (ex-2680d8)

```
> v :=  [ 6, -2, -7, 5, -2, -10, -3, 5, -7, -3, 10, -8, 5, 5, -8, 0 ];
> M := Matrix(4, 4, v); M;
[  6  -2  -7   5]
[ -2 -10  -3   5]
[ -7  -3  10  -8]
[  5   5  -8   0]
> Determinant(M); Factorization(Determinant(M));
1936
[ <2, 4>, <11, 2> ]
> HasseMinkowskiInvariant(M, 2);
-1
> HasseMinkowskiInvariant(M, 11);
1
> D := Orthogonalization(M); D; // signature (2,2)
[     6      0      0      0]
[     0    -96      0      0]
[     0      0     18      0]
[     0      0      0 -34848]
> pSignature(M, -1); // should be the difference of 2 and 2
0
> n := Degree(Parent(M));
> Q := Rationals();
> E := [ Q ! D[i][i] : i in [1..n]];
> &*[ &*[ HilbertSymbol(E[i], E[j], 2) : i in [j+1..n]] : j in [1..n-1]];
-1
> &*[ &*[ HilbertSymbol(E[i], E[j], 11) : i in [j+1..n]] : j in [1..n-1]];
1
> IsotropicSubspace(M);
RSpace of degree 4, dimension 2 over Integer Ring
Generators:
( 3 -3  0  4)
( 4 -4  0 -2)
Echelonized basis:
( 1 -1  0 16)
( 0  0  0 22)
> pSignature(M, 2) mod 8;
0
> pSignature(M, 11) mod 8;
4
> pSignature(M, 5) mod 8; // equals Dimension at good primes
4
```

Example: Witt Index of a Random Form (ex-ed2b1e)

```
> SetSeed(12345);
> n := 20;
> P := PolynomialRing(Integers(),n);
> f := &+[&+[Random([-10..10])*P.i*P.j : i in [j..n]] : j in [1..n]];
> M := ChangeRing(2*SymmetricMatrix(f), Integers()); M;
[12 -2 -8 1 -6 0 3 -5 -6 -5 10 7 -1 -3 -7 -5 -6 -3 -3 -8]
[-2 2 10 -10 3 6 -3 -9 0 -5 7 -5 -5 -4 1 -5 1 -6 7 -8]
[-8 10 14 1 6 -8 -3 4 -2 3 -4 7 0 -8 -6 -4 -5 7 -4 8]
[1 -10 1 6 -5 0 8 1 7 -1 7 -7 6 7 -1 -9 -8 1 6 6]
[-6 3 6 -5 12 7 -3 -10 -5 -4 -6 1 -5 -9 4 -8 2 5 -4 -10]
[0 6 -8 0 7 -6 -4 3 0 2 -3 0 4 10 5 -8 6 1 -7 2]
```

(continues on next page)

(continued from previous page)

```
[3 -3 -3 8 -3 -4 16 -5 -10 7 -9 9 -6 -2 0 -1 1 -4 6 2]
[-5 -9 4 1 -10 3 -5 2 -4 -3 0 -3 9 -1 3 -5 2 2 -1 -10]
[-6 0 -2 7 -5 0 -10 -4 0 -1 5 1 3 5 3 0 1 -7 5 10]
[-5 -5 3 -1 -4 2 7 -3 -1 10 -9 5 2 -4 8 6 -4 9 -3 5]
[10 7 -4 7 -6 -3 -9 0 5 -9 2 2 -7 3 8 -4 7 -3 -3 6]
[7 -5 7 -7 1 0 9 -3 1 5 2 6 -10 -6 0 -2 -3 8 2 -9]
[-1 -5 0 6 -5 4 -6 9 3 2 -7 -10 -10 7 0 8 -1 -2 9 3]
[-3 -4 -8 7 -9 10 -2 -1 5 -4 3 -6 7 -6 -2 -3 3 -9 9 6]
[-7 1 -6 -1 4 5 0 3 3 8 8 0 0 -2 2 -8 5 2 3 -4]
[-5 -5 -4 -9 -8 -8 -1 -5 0 6 -4 -2 8 -3 -8 -18 5 4 6 4]
[-6 1 -5 -8 2 6 1 2 1 -4 7 -3 -1 3 5 5 6 -3 7 -7]
[-3 -6 7 1 5 1 -4 2 -7 9 -3 8 -2 -9 2 4 -3 14 5 2]
[-3 7 -4 6 -4 -7 6 -1 5 -3 -3 2 9 9 3 6 7 5 16 -3]
[-8 -8 8 6 -10 2 2 -10 10 5 6 -9 3 6 -4 4 -7 2 -3 -4]
> D := Integers() ! Determinant(M); D;
276132003816103322711292
> [ <u[1], HasseInvariant(f, u[1])> : u in Factorization(D)];
[ <2, -1>, <3, -1>, <7, 1>, <199, 1>, <879089, -1>, <6263690372711, -1> ]
> &+[ pExcess(f, u[1]) : u in Factorization(D) cat [<-1, 0>]] mod 8;
0
> time S := IsotropicSubspace(f);
Time: 0.640
> Dimension(S);
8
> pSignature(f, -1); // difference of 12 and 8
4
> B := Basis(S);
> InnerProduct(B[1], B[1]*M);
0
> &and [InnerProduct(B[i], B[j]*M) eq 0 : i, j in [1..Dimension(S)]];
true
```

34.5 Equivalence

This section describes functionality to test equivalence of quadratic forms. Currently only implemented over the rationals.

IsRationallyEquivalent(X, Y): AlgMatElt, AlgMatElt -> BoolElt, AlgMatElt

Given two rational symmetric matrices X and Y, determine if they are rationally equivalent, and if so, return T with $Y = TXT^t$.

IsRationallyEquivalent(f, g): RngMPolElt, RngMPolElt -> BoolElt, AlgMatElt

Given two quadratic forms f and g over $\mathbb{Q}$, determine whether they are rationally equivalent, and if so, return a transform.

IsRationallySimilar(X, Y): AlgMatElt, AlgMatElt -> BoolElt, AlgMatElt, RngIntElt

Given two rational symmetric matrices X and Y, determine if they are rationally similar, and if so, return T and s with $sY = TXT^t$.

IsRationallySimilar(f, g): RngMPolElt, RngMPolElt -> BoolElt, AlgMatElt, RngIntElt

Given two quadratic forms f and g over $\mathbb{Q}$, determine whether they are rationally similar, and if so, return a transform and a similarity factor.

34.6 Bibliography

- [BS96] W. Bosma and P. Stevenhagen. On the computation of quadratic 2-class groups. *Journal de théorie des nombres de Bordeaux*, 8(2):283–313, 1996.
- [JC98] N.J.A. Sloane J.H. Conway. *Sphere Packings, Lattices and Groups.* Volume 290 of Grundlehren der Mathematischen Wissenschaften. Springer, New York–Berlin–Heidelberg, 3rd edition, 1998.
- [Lam05] T. Y. Lam. *Introduction to quadratic forms over fields.* Volume 67 of Graduate Studies in Mathematics. American Mathematical Society, Providence, RI, 2005. ISBN 0-8218-1095-2. URL: https://doi.org/10.1090/gsm/067, doi:10.1090/gsm/067.
- [Sim05a] Denis Simon. Quadratic equations in dimensions 4, 5 and more. Preprint, http://www.math.unicaen.fr/ simon/, 2005.

Chapter 35

Binary Quadratic Forms

35.1 Introduction

A binary quadratic form is a polynomial $ax^2 + bxy + cy^2$ with integer coefficients a, b, c. The form is primitive if the coefficients have no common factor. Only primitive forms are considered here. The discriminant of a form is $D = b^2 - 4ac$. For negative discriminants, only positive definite forms are considered, i.e. a and hence c must be positive. In Magma forms are printed as tuples $\langle a, b, c\rangle$.

Binary quadratic forms are related to many other topics in number theory, including quadratic fields, modular forms, and complex multiplication. Algorithms for binary quadratic forms provide efficient means of computing ideal class groups of orders in quadratic fields. By using the explicit relation of definite quadratic forms with lattices with nontrivial endomorphism ring in the complex plane, one can apply modular and elliptic functions to forms, and exploit the analytic theory of complex multiplication.

Quadratic forms of a given discriminant D correspond to ordered bases of ideals in an order in a quadratic number field, up to scaling by the rationals. For negative discriminants, the primitive reduced forms in this structure are in bijection with the class group of invertible ideals. For positive discriminants, the reduced orbits of forms are used for this purpose. Efficient algorithms are provided for composition, enumeration of reduced forms, class group computations, and discrete logarithms. The functions are implemented for quadratic forms for all discriminants, not only fundamental discriminants. Maps between forms of different discriminants are also provided.

The functionality for binary quadratic forms is rounded out with various functions for applying modular and elliptic functions to forms, and for class polynomials associated to class groups of definite forms.

35.2 Creation Functions

35.2.1 Creation of Structures

For any nonsquare integer D congruent to 0 or 1 modulo 4, binary quadratic forms of discriminant D may be created. The parent object of forms of discriminant D can be created using the following commands.

BinaryQuadraticForms(D): RngIntElt -> QuadBin

QuadraticForms(D): RngIntElt -> QuadBin

Create the structure of integral binary quadratic forms of discriminant D.

35.2.2 Creation of Forms

Binary quadratic forms may be created by coercing a triple $[a, b, c]$ of integer coefficients into the parent structure of forms of discriminant $D = b^2 - 4ac$. Other constructors are provided for constructing the group identity, prime forms, or allowing the omission of third element c of the sequence.

Identity(Q): QuadBin -> QuadBinElt

`Q ! 1: QuadBin, RngIntElt -> QuadBinElt`

Create the principal form in the structure Q of binary quadratic forms of discriminant D. The principal form is a reduced form equivalent to $X^2 - D/4Y^2$ when $D \equiv 0 \bmod 4$, or $X^2 + XY - (D-1)/4Y^2$ when $D \equiv 1 \bmod 4$.

`Q ! [a, b, c]: QuadBin, RngIntElt, RngIntElt, RngIntElt -> QuadBinElt`

`elt< Q | a, b, c>: QuadBin, RngIntElt, RngIntElt, RngIntElt -> QuadBinElt`

`elt< Q | a, b>: QuadBin, RngIntElt, RngIntElt -> QuadBinElt`

Returns the binary quadratic form $aX^2 + bXY + cY^2$ in the magma of forms Q of discriminant D. Here c is determined by the solution of the equality $D = b^2 - 4ac$; if no integer c exists satisfying this, an error will occur.

`PrimeForm(Q, p): QuadBin, RngIntElt -> QuadBinElt`

If p is a split prime or a ramified prime not dividing the conductor of the magma of quadratic forms Q, returns a quadratic form $pX^2 + bXY + cY^2$ in Q.

35.3 Basic Invariants

Structures of binary quadratic forms are defined in terms of a discriminant, and membership in a structure determined by this invariant. To aid in the construction of forms, additional elementary functions are provided to test integer inputs to determine if they define valid discriminants of quadratic forms.

`Discriminant(f): QuadBinElt -> RngIntElt`

The discriminant $b^2 - 4ac$ of a quadratic form $f = aX^2 + bXY + cY^2$.

`Discriminant(Q): QuadBin -> RngIntElt`

The discriminant of the quadratic forms in the structure Q.

`IsDiscriminant(D): RngIntElt -> BoolElt`

True iff the integer D is the discriminant of some quadratic form. This holds iff D is congruent to 0 or 1 mod 4.

`IsFundamental(D): RngIntElt -> BoolElt`

`IsFundamentalDiscriminant(D): RngIntElt -> BoolElt`

True iff the integer D is a fundamental discriminant, i.e. a discriminant that is not a square multiple of any smaller discriminant.

`FundamentalDiscriminant(D): RngIntElt -> RngIntElt`

The fundamental discriminant D_0 such that D is a square multiple of D_0.

`Conductor(Q): QuadBin -> RngIntElt`

Let D be `Discriminant(Q)`. The conductor is the integer m such that $D = m^2 D_0$, where D_0 is the fundamental discriminant.

35.4 Operations on Forms

35.4.1 Arithmetic

`Conjugate(f): QuadBinElt -> QuadBinElt`

Given a form $f = ax^2 + bxy + cy^2$, returns the conjugate form $ax^2 - bxy + cy^2$.

`f * g: QuadBinElt, QuadBinElt -> QuadBinElt`

`Composition(f, g): QuadBinElt, QuadBinElt -> QuadBinElt`

```
Al       : MonStgElt                    Default: "Gauss"
Reduction: BoolElt                      Default: false
```

Returns the composition of two binary quadratic forms f and g. The operator `*' returns a reduced form equivalent to the product using a fast composition algorithm of Shanks. In contrast, by default Composition returns the true composition in the group of forms, unless the parameter Reduction := true. The function Composition takes a further parameter Al which specifies whether the algorithm of Gauss or Shanks, set to "Gauss" by default. The algorithm of Shanks performs partial intermediate reductions, so it is not allowed to select Al := "Shanks" and Reduction := false.

f ^ n: QuadBinElt, RngIntElt -> QuadBinElt

Power(f, n): QuadBinElt, RngIntElt -> QuadBinElt

```
Al       : MonStgElt                    Default: "Gauss"
Reduction: BoolElt                      Default: false
```

Returns the n-th power of a form f. The operator `^' returns a reduced representative, using the fast composition algorithm of Shanks. In contrast, by default Power returns the true composition in the group of forms, unless the parameter Reduction := true. The function Power takes the further parameter Al which specifies whether the algorithm of Gauss or Shanks is used, set to "Gauss" by default. The algorithm of Shanks performs partial intermediate reductions, so it is not allowed to select Al := "Shanks" and Reduction := false.

35.4.2 Matrix Action

The right action of $\mathrm{SL}(2,\mathbb{Z})$ on the set of quadratic forms of discriminant D is given by the rule

$$f(x,y)\begin{pmatrix} r & s \\ t & u \end{pmatrix} = f(rx+sy, tx+uy).$$

f * M: QuadBinElt, GrpMatElt -> QuadBinElt

The right action of $M \in \mathrm{SL}(2,\mathbb{Z})$ on the binary quadratic form f.

35.4.3 Reduction

Reduction(f): QuadBinElt -> QuadBinElt, Mtrx

ReducedForm(f): QuadBinElt -> QuadBinElt, Mtrx

Returns a reduced quadratic form equivalent to f, and the transformation matrix.

ReductionStep(f): QuadBinElt -> QuadBinElt

The result of applying one reduction step to the quadratic form f.

ReductionOrbit(f): QuadBinElt -> {@ QuadBinElt @}

For a binary quadratic form f of positive discriminant, this returns an indexed set containing all reduced forms equivalent to f. These are obtained by iterating the ReductionStep operator, starting with Reduction(f).

Order(f): QuadBinElt -> RngIntElt

For a binary quadratic form f of discriminant D, this returns the order of f in the class group for discriminant D (see *ClassGroup*). Note that the class group is defined using the equivalence relation IsEquivalent(f1, f2 : Narrow := false).

35.4.4 Attribute Access

The coefficient sequence can be accessed as a sequence of integers, providing the inverse operation to the forms coercion constructor.

f[i]: QuadBinElt, RngIntElt -> RngIntElt

The i-th coefficient of f, where $1 \leq i \leq 3$.

Eltseq(f): QuadBinElt -> SeqEnum[RngIntElt]

`ElementToSequence(f): QuadBinElt -> SeqEnum[RngIntElt]`

The sequence $[a, b, c]$ where f is the form $ax^2 + bxy + cy^2$.

35.4.5 Boolean Operations

`f in Q: QuadBinElt, QuadBin -> BoolElt`

Return `true` if and only if f is in Q, that is f and Q have the same discriminant.

`f eq g: QuadBinElt, QuadBinElt -> BoolElt`

Return `true` if the quadratic form f and g are equal and `false` otherwise.

`IsIdentity(f): QuadBinElt -> BoolElt`

Return `true` if and only if f is the principal form in its parent structure.

`IsReduced(f): QuadBinElt -> BoolElt`

Return `true` if the quadratic form f is reduced; `false` otherwise.

`IsEquivalent(f, g): QuadBinElt, QuadBinElt -> BoolElt, AlgMatElt`

Narrow: BoolElt	Default: true

For binary quadratic forms f and g of the same discriminant, this returns whether the forms are equivalent; that is, whether they can be obtained from each other by a unimodular transformation. When the forms are equivalent, a transformation matrix is also returned.

If the parameter `Narrow` is `false`, the function tests a different notion of equivalence in the case of positive discriminant, which may be weaker. This is defined by merging the equivalence classes of forms $\langle a, b, c\rangle$ and $\langle -a, b, -c\rangle$ into a single equivalence class. (Another way to describe it is as follows: Forms are equivalent iff their associated ideals are in the same ordinary ideal class. On the other hand, "narrow" equivalence of forms corresponds to narrow ideal classes.) In this case, the transformation matrix may produce the complementary form of g.

35.4.6 Maps of Forms

`FundamentalQuotient(Q): QuadBin -> Map`

The quotient homomorphism from the class group of Q to the class group of fundamental discriminant.

`QuotientMap(Q1, Q2): QuadBin, QuadBin -> Map`

Given two structures of quadratic forms Q_1 and Q_2, such that the discriminant of Q_1 equals a square times the discriminant of Q_2, the quotient homomorphism from Q_1 to Q_2 is returned as a map.

`Q ! f: QuadBin, QuadBinElt -> QuadBinElt`

The `!` operator coerces a binary quadratic form f into the structure Q, when they are compatible. This requires that Q is `QuadraticForms(D)` and f has discriminant m^2D for some integer m.

35.4.7 Related Structures

`Parent(f): QuadBinElt -> QuadBin`

`Category(Q): QuadBin -> Cat`

`QuadraticOrder(Q): QuadBin -> RngQuad`

Given a structure of quadratic forms of discriminant D, returns the quadratic order of discriminant D.

`Ideal(f): QuadBinElt -> RngQuadIdl`

Given a quadratic form $f = ax^2 + bxy + cy^2$, returns the ideal $(a, (-b + \sqrt{D})/2)$ in the quadratic order $\mathbb{Z}[(t + \sqrt{D})/2]$, where t equals 0 or 1.

35.5 Reduced Forms

Note that in versions up to Magma V2.20, in cases where the narrow class group differs from the class group, these two functions behaved differently. In those cases, the `ReducedForms` and `ReducedOrbits` only covered half of the equivalence classes.

`ReducedForms(Q): QuadBin -> [ QuadBinElt ]`

`ReducedForms(D): RngIntElt -> [ QuadBinElt ]`

For Q = QuadraticForms(D), this returns a sequence containing precisely one reduced form in each equivalence class of primitive forms (under the usual equivalence relation, given by `IsEquivalent(f1, f2)` for forms `f1, f2`).

When $D < 0$, these are simply all the primitive reduced forms in `Q`.

`ReducedOrbits(Q): QuadBin -> [ {@ QuadBinElt @} ]`

For Q = QuadraticForms(D) where D is positive, this returns the `ReductionOrbit` of each of the `ReducedForms(Q)`, as a sequence of indexed sets.

35.6 Class Groups

In Magma, the class group of binary quadratic forms of discriminant D is defined by the equivalence relation `IsEquivalent(f1, f2 : Narrow := false)` for forms `f1, f2`. It is the same as the ideal class group (`PicardGroup`) of the quadratic order of discriminant D. The class group of binary forms for $D > 0$ is usually defined using standard ("narrow") equivalence of forms, which agrees with the narrow ideal class group of the order. For $D < 0$, there is no difference. For $D > 0$, the two class groups differ if and only if all units in the quadratic order have norm 1. When they are different, each class is the union of two narrow classes: forms $\langle a, b, c\rangle$ and $\langle -a, b, -c\rangle$ are equivalent but not narrowly equivalent.

`ClassNumber(Q: parameters): QuadBin -> RngIntElt`

`ClassNumber(D: parameters): RngIntElt -> RngIntElt`

```
Al              : MonStgElt                 Default: "Automatic"
FactorBasisBound: FldReElt                  Default: 0.1
ProofBound      : FldReElt                  Default: 6
ExtraRelations  : RngIntElt                 Default: 1
```

This returns the class number of binary quadratic forms Q of discriminant D.

The parameter `Al` may be supplied to select the method used to calculate the class number. The possible values are `"ReducedForms"` (enumerating all reduced forms), `"Shanks"` (using a Shanks-based algorithm in the class group), `"Sieve"`, or `"NoSieve"`. The default is to use reduced form enumeration for small discriminants, the Shanks algorithm for the middle range, a class group index-calculus for large discriminants and the sieve method described in [Jac99] for very large discriminants.

The parameters `FactorBasisBound`, `ProofBound`, and `ExtraRelations` are relevant only for the index calculus algorithm. They are described (below) under *ClassGroup*.

`ClassGroup(Q: parameters): QuadBin -> GrpAb, Map`

```
Al              : MonStgElt                 Default: "Automatic"
FactorBasisBound: FldReElt                  Default: 0.1
ProofBound      : FldReElt                  Default: 6
ExtraRelations  : RngIntElt                 Default: 1
```

This computes the class group of binary quadratic forms of discriminant D. The function returns the class group as an abelian group, together with a map from the group to quadratic forms.

Important: the class group is defined using non-strict equivalence (see the introduction to this subsection).

Depending on the size of D, either a Shanks-based method is used ([BJT97, Tes98a]), an index calculus variant ([CDO93, Coh93, HM89]) or a sieving method ([Jac99]). If the parameter `Al` is set to `"Sieve"` or D is larger than 10^{20}, then the sieving method is used. The results are only proven under GRH. If the parameter `Al` is set to `"NoSieve"` then the sieving method will not be used.

The parameters `FactorBasisBound`, `ProofBound`, and `ExtraRelations` apply to the index calculus method only. This method performs two steps. In the first step a factor basis containing prime forms of norm $< B_1$ is built. Next, one looks for generators for the full lattice of relations between the forms in the factor basis. The determinant of this lattice will be the class number and the Smith-form of the relation matrix gives the structure of the class group. Once the matrix is of full rank, the algorithm will look for `ExtraRelations` more relations to potentially decrease the discriminant. In the second step, for all prime forms of norm $< B_2$ it is verified that they are in the class group generated by the forms of the first step. The bounds are chosen to be $B_1 :=$ `FactorBasisBound`$\cdot \log^2 |D|$ and $B_2 :=$ `ProofBound`$\cdot \log^2 |D|$, so the result is correct under the assumption of GRH. The final result is then checked against the Euler product over the first 30000 primes. If the quotient becomes too large, a warning is issued. In this case one should increase the `ExtraRelations` parameter.

`ClassGroupStructure(Q: parameters): QuadBin -> [ RngIntElt ]`

The structure of the class group of the binary quadratic forms Q of discriminant D returned as a sequence of integers giving the abelian invariants.

The parameters are the same as for `ClassGroup`.

`AmbiguousForms(Q): QuadBin -> SeqEnum`

Enumerates the ambiguous forms of negative discriminant D, where D is the discriminant of the magma of binary quadratic forms Q.

`TwoTorsionSubgroup(Q): QuadBin -> GrpAb, Map`

The subgroup of 2–torsion elements of in the class group of Q.

Example: Computing in a Form Class Group (ex-d1a38f)

We give an example of some computations in the class group of a magma of quadratic forms. Elements in the class group are not really represented by single reduced forms but by cycles of equivalent reduced forms.

```
> Q<z> := QuadraticField(7537543);                        // arbitrary choice
> Q := QuadraticForms(Discriminant(Q));
> C, m := ClassGroup(Q);
> C;
Abelian Group isomorphic to Z/2 + Z/76
Defined on 2 generators
Relations:
    76*C.1 = 0
    2*C.2 = 0
>                                    // get the generators as quadratic forms:
> f := m(C.1);
> g := m(C.2);
> h := g^2;
> g, h;
<-1038,4894,1493> <-887,4340,3189>
> c := [];                       // create the cycle of forms equivalent to g^2:
> repeat
>     h := ReductionStep(h);
>     Append(~c, h);
> until h eq g^2;
> P := Parent(g);
> Identity(P) in c;
true                 // this proves that the second class has order dividing 2
```

(continues on next page)

(continued from previous page)

```
> for d in Divisors(76) do
>     c := [];
>     h := f^d;
>     repeat h := ReductionStep(h); Append(~c, h);
>     until h eq f^d;
>     d, Identity(P) in c, #c;
> end for;
1 false 16
2 false 14
4 false 12                          // the cycle lengths vary
19 false 18
38 false 16
76 true 14                          // so the true order of this class is 76
```

35.7 Discrete Logarithms

`Log(b, x): QuadBinElt, QuadBinElt -> RngIntElt`

The discrete logarithm of binary quadratic form x with respect to base b, or -1 if Magma can that determine no solution exists. This function exists only for negative discriminant forms. Computation is done via the Pohlig-Hellman algorithm along with a collision search subroutine (a variant of Pollard's rho method). If the user is unsure whether a solution exists, it is safest to use `Log` with a time limit (see below) to prevent an infinite loop in the collision search.

`Log(b, x, t): QuadBinElt, QuadBinElt, RngIntElt -> RngIntElt`

Searches for up to t seconds for the discrete logarithm of binary quadratic form x with respect to base b. This function exists only for negative discriminant forms. If Magma is able to determine no solution exists, then -1 will be returned. If no solution is found within the given time frame, then -2 will be returned. Computation is done via the Pohlig-Hellman algorithm along with a collision search subroutine (a variant of Pollard's rho method).

35.8 Elliptic and Modular Invariants

Binary quadratic forms of negative discriminant describe positive definite lattices in the complex plane, with integral-valued inner product. As such, it is possible to apply modular and elliptic functions to the form, interpreting this as an element of the upper half plane.

`Lattice(f): QuadBinElt -> Lat`

Given a binary quadratic form $f = ax^2 + bxy + cy^2$ of negative discriminant, returns the rank two lattice of f having Gram matrix

$$\begin{pmatrix} a & b/2 \\ b/2 & c \end{pmatrix}.$$

Note that the lattice L is the half-integral lattice such that integral representations $f(x, y) = n$ are in bijection with vectors (x, y) of norm n, which will be a rational number.

`GramMatrix(f): QuadBinElt -> AlgMatElt`

Returns the Gram matrix of the binary quadratic form f, which need not be of negative discriminant. The matrix will be half-integral and defined over the rationals.

`ThetaSeries(f, n): QuadBinElt, RngIntElt -> RngSerElt`

The integral theta series of the binary quadratic form f to precision n.

`RepresentationNumber(f, n): QuadBinElt, RngIntElt -> RngIntElt`

The nth representation number of the form f of negative discriminant.

`jInvariant(f): QuadBinElt -> RngSerElt`

For a binary quadratic form $f = ax^2 + bxy + cy^2$ with negative discriminant, return the j–invariant of f, equal to the j–invariant of $\tau = (-b + \sqrt{b^2 - 4ac})/2a$.

`Eisenstein(k, f): RngIntElt, QuadBinElt -> RngSerElt`

Given a positive even integer $k = 2n$ and a binary quadratic form $f = ax^2 + bxy + cy^2$, return the value of the Eisenstein series $E_k(L)$ at the complex lattice $L = \langle a, (-b + \sqrt{b^2 - 4ac})/2 \rangle$.

`WeierstrassSeries(z, f): RngSerElt, QuadBinElt -> RngSerElt`

Given a complex power series z with positive valuation and a binary quadratic form $f = ax^2 + bxy + cy^2$, returns the q–expansion of the Weierstrass $\wp$-function at the complex lattice $L = \langle a, (-b + \sqrt{b^2 - 4ac})/2 \rangle$.

Example: Elliptic and Modular Invariants of a Form `(ex-f93e36)`

```
> Q := QuadraticForms(-163);
> f := PrimeForm(Q,41);
> CC<i> := ComplexField();
> PC<z> := LaurentSeriesRing(CC);
> x := WeierstrassSeries(z,f);
> y := -Derivative(x)/2;
> A := -Eisenstein(4,f)/48;
> B := Eisenstein(6,f)/864;
> Evaluate(y^2 - (x^3 + A*x + B),1/2);
1.384608660824596881000000000 E-26 - 1.305091481190174818000000000 E-26*i
```

35.9 Class Invariants

There are intrinsics `HilbertClassPolynomial(D): RngIntElt → RngUPolElt` and `WeberClassPolynomial(D): RngIntElt → RngUPolElt, FldFunRatUElt` to compute the minimal polynomial over $\mathbb{Q}$ of the *j-invariant* associated to the imaginary quadratic order of discriminant D or the smaller (in terms of coefficient size) minimal polynomial of an associated Weber function. These are described in detail in Section Class Polynomials of the Modular Curves chapter.

35.10 Bibliography

- [BJT97] J. Buchmann, M. J. Jacobson, Jr., and E. Teske. On Some Computational Problems in Finite Abelian Groups. *Mathematics of Computation*, 66:1663–1687, 1997.
- [CDO93] H. Cohen, F. Diaz y Diaz, and M. Olivier. Calculs de nombres de classes et de régulateurs de corps quadratiques en temps sous-exponentiel. In *Séeminaire de Théorie des Nombres, Paris, 1990–91*, volume 108 of Progr. Math., pages 35–46. Birkhäuser Boston, Boston, MA, 1993.
- [Coh93] Henri Cohen. *A Course in Computational Algebraic Number Theory.* Volume 138 of Graduate Texts in Mathematics. Springer, Berlin–Heidelberg–New York, 1993.
- [HM89] J. Hafner and K. McCurley. A rigorous subexponential algorithm for computation of class groups. *Jornal American Math. Soc.*, 2:837 – 850, 1989.
- [Tes98a] E. Teske. A Space Efficient Algorithm for Group Structure Computation. *Mathematics of Computation*, 67:1637–1663, 1998.
- [Jac99] M. J. Jacobson, Jr. Applying sieving to the computation of quadratic class groups. *Math. Comp.*, 68(226):859–867, 1999.

References

[AEVZ02] E. Agrell, T. Eriksson, A. Vardy, and K. Zeger. Closest point search in lattices. *IEEE Transactions on Information Theory*, 48(8):2201–2214, 2002.

[Ajt98] Miklós Ajtai. The Shortest Vector Problem in L2 is NP-hard for Randomized Reductions (Extended Abstract). In *Proceedings of the 30th Symposium on the Theory of Computing (STOC 1998)*, 10–19. ACM, 1998.

[Akh02] Ali Akhavi. Random lattices, threshold phenomena and efficient reduction algorithms. *Theoretical Computer Science*, 287(2):359–385, 2002.

[Bel03] C. N. Beli. Integral spinor norm groups over dyadic local fields. *J. Number Theory*, 102(1):125–182, 2003. URL: http://dx.doi.org/10.1016/S0022-314X(03)00057-X, doi:10.1016/S0022-314X(03)00057-X.

[BS96] W. Bosma and P. Stevenhagen. On the computation of quadratic 2-class groups. *Journal de théorie des nombres de Bordeaux*, 8(2):283–313, 1996.

[BJT97] J. Buchmann, M. J. Jacobson, Jr., and E. Teske. On Some Computational Problems in Finite Abelian Groups. *Mathematics of Computation*, 66:1663–1687, 1997.

[Cho15] S. Cho. Group schemes and local densities of quadratic lattices in residue characteristic 2. *Compos. Math.*, 151(5):793–827, 2015. URL: http://dx.doi.org/10.1112/S0010437X14007829, doi:10.1112/S0010437X14007829.

[CDO93] H. Cohen, F. Diaz y Diaz, and M. Olivier. Calculs de nombres de classes et de régulateurs de corps quadratiques en temps sous-exponentiel. In *Séeminaire de Théorie des Nombres, Paris, 1990–91*, volume 108 of Progr. Math., pages 35–46. Birkhäuser Boston, Boston, MA, 1993.

[Coh93] Henri Cohen. *A Course in Computational Algebraic Number Theory*. Volume 138 of Graduate Texts in Mathematics. Springer, Berlin–Heidelberg–New York, 1993.

[CH87] P. J. Costello and J. S. Hsia. Even Unimodular 12-Dimensional Quadratic Forms over $\mathbf{Q}(\sqrt{5})$. *Adv. Math.*, 64:241–278, 1987.

[dW87] Benne M.M. de Weger. Solving exponential Diophantine equations using lattice basis reduction algorithms. *J. Number Th.*, 26:325–367, 1987.

[FP83] U. Fincke and M. Pohst. A procedure for determining algebraic integers of given norm. In *EUROCAL*, volume 162 of LNCS, 194–202. Springer, 1983.

[GY00] W. T. Gan and J.-K. Yu. Group schemes and local densities. *Duke Math. J.*, 105(3):497–524, 2000. URL: http://dx.doi.org/10.1215/S0012-7094-00-10535-2, doi:10.1215/S0012-7094-00-10535-2.

[GHY01] Wee Teck Gan, Jonathan P. Hanke, and Jiu-Kang Yu. On an exact mass formula of Shimura. *Duke Math. J.*, 107(1):103–133, 2001. URL: http://dx.doi.org/10.1215/S0012-7094-01-10716-3, doi:10.1215/S0012-7094-01-10716-3.

[Ger03] Larry J. Gerstein. Definite quadratic forms over $\mathbb{F}_q[X]$. *J. Algebra*, 268(1):252–263, 2003.

[HM89] J. Hafner and K. McCurley. A rigorous subexponential algorithm for computation of class groups. *Jornal American Math. Soc.*, 2:837 – 850, 1989.

[HS07] Guillaume Hanrot and Damien Stehlé. Improved Analysis of Kannan's Shortest Lattice Vector Algorithm (Extended Abstract). In *Advances in cryptology—CRYPTO 2007*, volume 4622 of LNCS, 170–186. Springer, 2007.

[JC98] N.J.A. Sloane J.H. Conway. *Sphere Packings, Lattices and Groups*. Volume 290 of Grundlehren der Mathematischen Wissenschaften. Springer, New York–Berlin–Heidelberg, 3rd edition, 1998.

[Jac62a] Ronald Jacobowitz. Hermitian forms over local fields. *Amer. J. Math.*, 84:441–465, 1962. URL: https://doi.org/10.2307/2372982, doi:10.2307/2372982.

[Kan83] R. Kannan. Improved algorithms for integer programming and related lattice problems. In *Proceedings of the 15th Symposium on the Theory of Computing (STOC 1983)*, 99–108. ACM, 1983.

[Kir12] M. Kirschmer. A normal form for definite quadratic forms over $\mathbb{F}_q[t]$. *Math. Comp.*, 81:1619–1634, 2012.

[Kir16] M. Kirschmer. *Definite quadratic and hermitian form with small class number*. Habilitation, RWTH Aachen University, 2016.

[Kne57] M. Kneser. Klassenzahlen indefiniter quadratischer Formen in drei oder mehr Veränderlichen. *Archiv Math.*, 8:241–250, 1957.

[Lam05] T. Y. Lam. *Introduction to quadratic forms over fields*. Volume 67 of Graduate Studies in Mathematics. American Mathematical Society, Providence, RI, 2005. ISBN 0-8218-1095-2. URL: https://doi.org/10.1090/gsm/067, doi:10.1090/gsm/067.

[LLL82] Arjen K. Lenstra, Hendrik W. Lenstra, and László Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261:515–534, 1982.

[Lor16] D. Lorch. *Einklassige Geschlechter orthogonaler Gruppen*. PhD thesis, RWTH Aachen University, 2016.

[Maa41] H. Maass. Modulformen und quadratische Formen über dem quadratischen Zahlkörper $R(\sqrt{5})$. *Math. Ann.*, 118:65–84, 1941.

[MG02] Daniele Micciancio and Shafi Goldwasser. *Complexity of lattice problems: a cryptographic perspective*. Volume 671 of The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, 2002. ISBN 0-7923-7688-9.

[NS01a] G. Nebe and N.J.A. Sloane. The Catalogue of Lattices. http://www.research.att.com/∼njas/lattices/, 2001.

[NS01b] Gabriele Nebe and Neil J.A. Sloane. A Catalogue of Lattices. http://akpublic.research.att.com/∼njas/lattices/index.html, 2001.

[NS] P. Nguyen, D. Stehlé. “LLL on the Average.” 238–256.

[NS09a] Phong Nguyen and Damien Stehlé. An LLL Algorithm with Quadratic Complexity. *SIAM Journal on Computing*, 39(3):874–903, 2009.

[NS09c] Phong Q. Nguyen and Damien Stehlé. Low-dimensional lattice basis reduction revisited. *ACM Trans. Algorithms*, 5(4):Art. 46, 48, 2009. URL: https://doi.org/10.1145/1597036.1597050, doi:10.1145/1597036.1597050.

[OMe73] O. T. O'Meara. *Introduction to Quadratic Forms*. Springer, 1973.

[Pau98] Sachar Paulus. Lattice Basis Reduction in Function Fields. In *In ANTS-3 : Algorithmic*, 567–575. Springer-Verlag, 1998.

[Ple74] Wilhelm Plesken. *Beiträge zur Bestimmung der endlichen irreduziblen Untergruppen von GL(n,Z) und ihrer ganzzahligen Darstellungen*. PhD thesis, RWTH Aachen, 1974.

[PS97a] Wilhelm Plesken and Bernd Souvignier. Computing Isometries of Lattices. *J. Symbolic Comp.*, 24(3):327–334, 1997.

[Poh87] Michael Pohst. A Modification of the LLL Reduction Algorithm. *J. Symbolic Comp.*, 4(1):123–127, 1987.

[Pro] The SPACES Project. MPFR, a LGPL-library for multiple-precision floating-point computations with exact rounding. http://www.mpfr.org/.

[PS08] Xavier Pujol and Damien Stehlé. Rigorous and efficient short lattice vectors enumeration. In *Advances in Cryptology—AsiaCrypt 2008*, LNCS. Springer, 2008.

[SE94] Claus-Peter Schnorr and Michael Euchner. Lattice Basis Reduction: Improved Practical Algorithms and Solving Subset Sum Problems. *Mathematics of Programming*, 66:181–199, 1994.

[SH95] Claus-Peter Schnorr and Horst Helmut Hörner. Attacking the Chor-Rivest Cryptosystem by Improved Lattice Reduction. In *Advances in Cryptology—EuroCrypt 1995*, volume 921 of LNCS, 1–12. Springer-Verlag, 1995.

[SP91] Rainer Schulze-Pillot. An algorithm for computing genera of ternary and quaternary quadratic forms. In Stephen M. Watt, editor, *Proceedings ISSAC'91*, 134–143. Bonn, 1991.

[Shi99] G. Shimura. An exact mass formula for orthogonal groups. *Duke Math. J.*, 97(1):1–66, 1999. URL: http://dx.doi.org/10.1215/S0012-7094-99-09701-6, doi:10.1215/S0012-7094-99-09701-6.

[Sho] Victor Shoup. NTL, Number Theory C++ Library. http://www.shoup.net/ntl/.

[Sie35] C. L. Siegel. Über die analytische Theorie der quadratischen Formen. *Ann. of Math. (2)*, 36(3):527–606, 1935. URL: http://dx.doi.org/10.2307/1968644, doi:10.2307/1968644.

[Sim05a] Denis Simon. Quadratic equations in dimensions 4, 5 and more. Preprint, http://www.math.unicaen.fr/ simon/, 2005.

[Sim05b] Denis Simon. Solving quadratic equations using reduced unimodular quadratic forms. *Math. Comp.*, 74(251):1531–1543 (electronic), 2005.

[Ste09] Damien Stehlé. *Floating-point LLL: theoretical and practical aspects*. Springer-Verlag, 2009, To appear.

[Tes98a] E. Teske. A Space Efficient Algorithm for Group Structure Computation. *Mathematics of Computation*, 67:1637–1663, 1998.

[vEB81] Peter van Emde Boas. Another NP-complete partition problem and the complexity of computing short vectors in a lattice. Technical report 81-04, Mathematisch Instituut, Universiteit van Amsterdam, 1981.

[Jac99] M. J. Jacobson, Jr. Applying sieving to the computation of quadratic class groups. *Math. Comp.*, 68(226):859–867, 1999.

Index

Symbols

(v, w) (*literal*), 10
D (*operation*), 57
G (*operation*), 51, 52
V (*operation*), 102
- v (*operation*), 9, 79

A

A + B (*operation*), 65
A eq B (*operation*), 76
A meet B (*operation*), 65
A ne B (*operation*), 76
ActionGenerator(L, i) (*function*), 96
AdjacencyMatrix(G, p) (*function*), 54
Advance(nProc) (*procedure*), 75
AmbientSpace(L) (*function*), 12, 69
AmbiguousForms(Q) (*function*), 118
AntisymmetricForms(L) (*function*), 97
AntisymmetricForms(L, n) (*function*), 97
Arithmetic of Number Field Lattice Elements (*example*), 79
Attributes of Number Field Lattices (*example*), 74
Automorphism Group of a 19-Dimensional Lattice (*example*), 92
AutomorphismGroup(F) (*function*), 91
AutomorphismGroup(G) (*function*), 95
AutomorphismGroup(L) (*function*), 77, 89
AutomorphismGroup(L, F) (*function*), 91
AutomorphismGroup(L, v) (*function*), 86
Automorphisms of E8 Acting on Vectors (*example*), 91
AutomorphousClasses(G, p) (*function*), 51
AutomorphousClasses(L, p) (*function*), 51
AuxiliaryForms(L) (*function*), 13, 70

B

BadPrimes(L) (*function*), 13, 70
Barnes-Wall Lattice from a Reed-Muller Code (*example*), 6
BaseChange(L, S) (*function*), 14
BaseExtend(L, S) (*function*), 14
BaseRing(L) (*function*), 14, 69
Basic Number Field Lattices (*example*), 81
Basis(L) (*function*), 13, 68
BasisDenominator(L) (*function*), 13
BasisMatrix(e) (*function*), 103
BasisMatrix(L) (*function*), 13, 68
BasisReduction(L) (*function*), 24
BasisReduction(X) (*function*), 23
BasisScaling(L, r) (*function*), 65
BinaryQuadraticForms(D) (*function*), 113
BKZ Reduction of a Random Basis (*example*), 29
BKZ(L, n) (*function*), 29
BKZ(M, n) (*function*), 29

C

CanonicalLocalGenus(G) (*function*), 50
Category(L) (*function*), 12
Category(Q) (*function*), 116
CenterDensity(L) (*function*), 32
CenterDensity(L, K) (*function*), 32
CentralEndomorphisms(L, n) (*function*), 97
CentreDensity(L) (*function*), 32
CentreDensity(L, K) (*function*), 32
CentreOfEndomorphismRing(L) (*function*), 97
ChangeRing(L, R) (*function*), 65
ChangeRing(L, S) (*function*), 14
Cholesky(L) (*function*), 48
Cholesky(L, K) (*function*), 48
Cholesky(M) (*function*), 48
Cholesky(M, K) (*function*), 48
ClassGroup(Q: parameters) (*function*), 117
ClassGroupStructure(Q: parameters) (*function*), 118
ClassNumber(D: parameters) (*function*), 117
ClassNumber(Q: parameters) (*function*), 117
ClosestVector(L, w) (*function*), 34
ClosestVectors(L, w) (*function*), 33
ClosestVectorsMatrix(L, w) (*function*), 33
CloseVector(L, w, l, u) (*function*), 36
CloseVector(L, w, u) (*function*), 36
CloseVectors(L, w, l, u) (*function*), 35
CloseVectors(L, w, u) (*function*), 35
CloseVectorsMatrix(L, l, u) (*function*), 36
CloseVectorsMatrix(L, w, l, u) (*function*), 36
CloseVectorsMatrix(L, w, u) (*function*), 36
CloseVectorsProcess(L, w, l, u) (*function*), 40
CloseVectorsProcess(L, w, u) (*function*), 40
CoefficientIdeals(L) (*function*), 13, 68
CoefficientRing(L) (*function*), 14
Compartments(G) (*function*), 52
Composition(f, g) (*function*), 114
Computing a Theta Series (*example*), 41
Computing in a Form Class Group (*example*), 118
Conductor(Q) (*function*), 114

Conjugate(f) (*function*), 114
Constituents(V) (*function*), 103
Constructing E8 as a 2-Neighbour (*example*), 54
Content(L) (*function*), 12
ConwaySymbol(G) (*function*), 51, 52
Coordelt(L, C) (*function*), 9
CoordinateLattice(L) (*function*), 4
CoordinateRing(L) (*function*), 14, 69
Coordinates(L, v) (*function*), 11
Coordinates(v) (*function*), 11, 81
CoordinateSpace(L) (*function*), 12
CoordinatesToElement(L, C) (*function*), 9
CoordinatesToLattice(L, S) (*function*), 78
CoordinatesToLattice(L, v) (*function*), 78
CoordinateVector(L, v) (*function*), 11
CoordinateVector(v) (*function*), 11
CoveringRadius(L) (*function*), 46
Creating a Lattice in Four Ways (*example*), 4
Creating a Lattice of Sublattices (*example*), 102
Creating Number Field Lattices (*example*), 67

D

DeepHoles(L) (*function*), 46
Degree(L) (*function*), 12, 69
Degree(v) (*function*), 12
Density(L) (*function*), 32
Density(L, K) (*function*), 32
Determinant(G) (*function*), 51, 52
Determinant(L) (*function*), 12, 69
Dimension(G) (*function*), 51, 52
Dimension(L) (*function*), 12, 69
DimensionOfCentreOfEndomorphismRing(L) (*function*), 98
DimensionOfEndomorphismRing(L) (*function*), 97
DirectSum(A) (*function*), 66
DirectSum(A, B) (*function*), 66
DirectSum(L, M) (*function*), 17
Discriminant(A, B) (*function*), 16, 70
Discriminant(f) (*function*), 114
Discriminant(L) (*function*), 12, 69
Discriminant(Q) (*function*), 114
DominantDiagonalForm(X) (*function*), 94
Dual Quotient of a Lattice (*example*), 17
Dual(L) (*function*), 16, 66
DualBasisLattice(L) (*function*), 17
DualQuotient(L) (*function*), 17

E

e + f (*operation*), 103
e eq f (*operation*), 103
e meet f (*operation*), 103
Eisenstein(k, f) (*function*), 120
ElementaryDivisors(A, B) (*function*), 15, 70
ElementToSequence(f) (*function*), 115
ElementToSequence(v) (*function*), 11
Elliptic and Modular Invariants of a Form (*example*), 120
elt< Q | a, b, c> (*constructor*), 114
elt< Q | a, b> (*constructor*), 114
elt<L | Q> (*constructor*), 9
Eltseq(f) (*function*), 115
Eltseq(v) (*function*), 11, 80
EmbeddingSpace(L) (*function*), 69
EndomorphismRing(L) (*function*), 97
Endomorphisms(L, n) (*function*), 97
EnumerationCost(L) (*function*), 42
EnumerationCost(L, u) (*function*), 42
EnumerationCostArray(L) (*function*), 43
EnumerationCostArray(L, u) (*function*), 43
Estimating Enumeration Cost (*example*), 43
EvenSublattice(L) (*function*), 17
Excess(G) (*function*), 52
ExpandBasis(M) (*function*), 59
ext< L | S > (*constructor*), 15
ext<L | RHS> (*constructor*), 65
Extended GCD via LLL (*example*), 24
ExteriorSquare(L) (*function*), 18

F

f * g (*operation*), 114
f * M (*operation*), 115
f ^ n (*operation*), 115
f eq g (*operation*), 116
f in Q (*operation*), 116
FreeBasis(L) (*function*), 68
FundamentalDiscriminant(D) (*function*), 114
FundamentalQuotient(Q) (*function*), 116
f[i] (*indexed*), 115

G

G1 eq G2 (*operation*), 51, 52
GaussReduce(L) (*function*), 28
GaussReduce(X) (*function*), 28
GaussReduceGram(F) (*function*), 28
Genera(r, s, d) (*function*), 49
GeneratorMatrix(L) (*function*), 68
Generators(L) (*function*), 68
Genus of the Coxeter-Todd Lattice (*example*), 54
Genus(G) (*function*), 49
Genus(L) (*function*), 49, 71
Genus(r, s, d, S) (*function*), 50
GenusRepresentatives(L) (*function*), 53, 73
GenusSymbol(L, p) (*function*), 70
GoodBasisOfNormGenerators(L, p) (*function*), 70
Gram Matrices and Transformations (*example*), 81
GramMatrix(f) (*function*), 119
GramMatrix(G) (*function*), 52
GramMatrix(L) (*function*), 13, 69, 107
GramMatrix(L, S) (*function*), 13, 69
GramMatrix(S) (*function*), 13, 69

GramMatrix(X) (*function*), 13
GreedyOrbit(M) (*function*), 30
GreedyReduction(L) (*function*), 30
Group(L) (*function*), 96

H

HasseInvariant(f, p) (*function*), 108
HasseInvariant(L, P) (*function*), 108
HasseInvariant(L, p) (*function*), 71, 108
HasseInvariant(M, p) (*function*), 71, 108
HasseInvariants(f) (*function*), 109
HasseInvariants(L) (*function*), 109
HasseInvariants(M) (*function*), 109
HasseMinkowskiInvariant(f, p) (*function*), 108
HasseMinkowskiInvariant(L, P) (*function*), 108
HasseMinkowskiInvariant(L, p) (*function*), 108
HasseMinkowskiInvariant(M, p) (*function*), 108
HasseMinkowskiInvariants(f) (*function*), 109
HasseMinkowskiInvariants(L) (*function*), 109
HasseMinkowskiInvariants(M) (*function*), 109
HermiteConstant(n) (*function*), 31
HermiteNumber(L) (*function*), 32
Hermitian Coxeter-Todd Lattice (*example*), 59
HermitianAutomorphismGroup(M) (*function*), 59
HermitianTranspose(M) (*function*), 59
HKZ Reduction (*example*), 28
HKZ(L) (*function*), 28
HKZ(X) (*function*), 27
HKZGram(F) (*function*), 28
Holes(L) (*function*), 46

I

Ideal(f) (*function*), 116
Identity(Q) (*function*), 113
Index(A, B) (*function*), 70
Index(L, S) (*function*), 16
InnerProduct(v, w) (*function*), 10, 79
InnerProductMatrix(L) (*function*), 13, 69
InnerProductScaling(L, r) (*function*), 65
IntegerRing() ! e (*operation*), 103
IntegralBasisLattice(L) (*function*), 18
Invariant Lattices of the Root Lattice A5 (*example*), 103
InvariantForms(G) (*function*), 59
InvariantForms(L) (*function*), 96
InvariantForms(L, n) (*function*), 97
Invariants of the Leech Lattice (*example*), 32
Involution(L) (*function*), 13, 68
IsDefinite(L) (*function*), 75
IsDiscriminant(D) (*function*), 114
IsEmpty(P) (*function*), 40
IsEquivalent(f, g) (*function*), 116
IsEven(G) (*function*), 51, 52
IsEven(L) (*function*), 14, 76
IsExact(L) (*function*), 14
IsFree(L) (*function*), 75
IsFull(L) (*function*), 14, 75
IsFundamental(D) (*function*), 114
IsFundamentalDiscriminant(D) (*function*), 114
IsGenus(G) (*function*), 51
IsGLattice(L) (*function*), 96
IsHermitian(L) (*function*), 14, 75
IsIdentical(A, B) (*function*), 76
IsIdentity(f) (*function*), 116
IsIntegral(L) (*function*), 14, 76
IsIntegral(L, p) (*function*), 76
IsIsometric(A, B) (*function*), 77
IsIsometric(F_1, F_2) (*function*), 93
IsIsometric(G1, G2) (*function*), 95
IsIsometric(L, F_1, M, F_2) (*function*), 93
IsIsometric(L, M) (*function*), 92, 93
IsIsometric(L, v, w) (*function*), 87
IsIsomorphic(F_1, F_2) (*function*), 93
IsIsomorphic(L, F_1, M, F_2) (*function*), 93
IsIsomorphic(L, M) (*function*), 92, 93
IsIsotropic(L, p) (*function*), 76
IsLocallyIsometric(L1, L2, p) (*function*), 77
IsLorentzian(L) (*function*), 86
IsMaximal(L) (*function*), 76
IsMaximal(L, p) (*function*), 76
IsMaximalIntegral(L) (*function*), 76
IsMaximalIntegral(L, p) (*function*), 76
IsModular(L) (*function*), 76
IsModular(L, p) (*function*), 76
IsNegativeDefinite(F) (*function*), 49
IsNegativeSemiDefinite(F) (*function*), 49
Isometries of a Lorentzian Lattice (*example*), 87
Isometry of Barnes-Wall Lattices (*example*), 93
Isometry Testing over a Polynomial Ring (*example*), 94
IsotropicSubspace(f) (*function*), 109
IsotropicSubspace(M) (*function*), 109
Isotropy and Witt Index of a Form (*example*), 109
IsPositiveDefinite(F) (*function*), 49
IsPositiveSemiDefinite(F) (*function*), 49
IsProperlyIsometric(A, B) (*function*), 77
IsProperlyIsometric(L, M) (*function*), 92
IsQuadratic(L) (*function*), 14, 75
IsRationallyEquivalent(f, g) (*function*), 111
IsRationallyEquivalent(L1, L2) (*function*), 77
IsRationallyEquivalent(L1, L2, p) (*function*), 77
IsRationallyEquivalent(X, Y) (*function*), 111
IsRationallySimilar(f, g) (*function*), 111
IsRationallySimilar(X, Y) (*function*), 111
IsReduced(f) (*function*), 116
IsRepresented(L, e) (*function*), 78
IsSameGenus(L1, L2) (*function*), 77
IsSimilar(A, B) (*function*), 78
IsSimple(L) (*function*), 75
IsSpacelike(v) (*function*), 86
IsSpinorGenus(G) (*function*), 51

IsSpinorNorm(G, p) (*function*), 51
IsSublattice(S, L) (*function*), 76
IsTimelike(v) (*function*), 86
IsTotallyPositiveDefinite(L) (*function*), 75
IsZero(L) (*function*), 14, 75
IsZero(v) (*function*), 10, 79
IteratedNeighbours(L, p) (*function*), 73

J

J * L (*operation*), 15, 65
jInvariant(f) (*function*), 119
JordanDecomposition(L, P) (*function*), 50
JordanDecomposition(L, p) (*function*), 50, 70

K

KissingNumber(L) (*function*), 32

L

L ! 0 (*operation*), 9
L ! e (*operation*), 78
L ! Q (*operation*), 9
L * J (*operation*), 65
L * r (*operation*), 65
L * s (*operation*), 15
L * T (*operation*), 66
L + M (*operation*), 17
L . i (*operation*), 9, 78
L / J (*operation*), 65
L / r (*operation*), 65
L / S (*operation*), 15
L / s (*operation*), 15
L eq M (*operation*), 13
L meet M (*operation*), 17
L ne M (*operation*), 13
L subset M (*operation*), 14
L`Minimum (*attribute*), 56
L`MinimumBound (*attribute*), 56
LargestDimension(D) (*function*), 57
Lattice of an Equation Order (*example*), 7
Lattice(C, "A") (*function*), 6
Lattice(C, "B") (*function*), 6
Lattice(D, d, i: parameters) (*function*), 58
Lattice(D, i: parameters) (*function*), 58
Lattice(D, N, i: parameters) (*function*), 58
Lattice(D, N: parameters) (*function*), 58
Lattice(e) (*function*), 103
Lattice(f) (*function*), 119
Lattice(G) (*function*), 96
Lattice(I) (*function*), 7
Lattice(n, Q) (*function*), 2
Lattice(n, Q, M) (*function*), 2
Lattice(O) (*function*), 7
Lattice(S) (*function*), 2
Lattice(S, M) (*function*), 2
Lattice(V, L) (*function*), 65
Lattice(X) (*function*), 2
Lattice(X, M) (*function*), 2
Lattice(X, n) (*function*), 8
LatticeData(D, d, i) (*function*), 58
LatticeData(D, i) (*function*), 58
LatticeData(D, N) (*function*), 58
LatticeData(D, N, i) (*function*), 58
LatticeDatabase() (*function*), 57
LatticeName(D, d, i) (*function*), 57
LatticeName(D, i) (*function*), 57
LatticeName(D, N) (*function*), 57
LatticeName(D, N, i) (*function*), 57
Lattices Invariant under SL(2,7) (*example*), 104
Lattices Related to the Icosahedron (*example*), 82
LatticeWithBasis(B) (*function*), 3
LatticeWithBasis(B, M) (*function*), 3
LatticeWithBasis(G, B) (*function*), 96
LatticeWithBasis(G, B, M) (*function*), 96
LatticeWithBasis(n, Q) (*function*), 3
LatticeWithBasis(n, Q, M) (*function*), 3
LatticeWithBasis(S) (*function*), 3
LatticeWithBasis(S, M) (*function*), 3
LatticeWithBasis(V, B) (*function*), 64
LatticeWithBasis(V, B, J) (*function*), 64
LatticeWithGram(F) (*function*), 3
LatticeWithGram(G, F) (*function*), 96
LatticeWithGram(n, Q) (*function*), 3
LatticeWithPseudobasis(V, P) (*function*), 65
Length(v) (*function*), 10
Length(v, K) (*function*), 10
Level(L) (*function*), 12
Level(V, i) (*function*), 102
Levels(v) (*function*), 103
LLL(L) (*function*), 24
LLL(X) (*function*), 21
LLLBasisMatrix(L) (*function*), 24
LLLGram(F) (*function*), 23
LLLGramMatrix(L) (*function*), 24
LocalBasis(L, p) (*function*), 68
LocalFactor(L, p) (*function*), 72
LocalGenera(G) (*function*), 51
LocalGenera(n, d, p) (*function*), 50
LocalGenus(A, p) (*function*), 50
LocalGenus(L, p) (*function*), 50, 71
LocalGenus(p, v, r, d) (*function*), 50
LocalModification(L, G, p) (*function*), 50
Log(b, x) (*function*), 119
Log(b, x, t) (*function*), 119
Looking Up a Lattice in the Database (*example*), 58

M

MakeAmbientInnerProduct(~L, IP) (*procedure*), 69
Mass(L) (*function*), 72
MaximalIntegralLattice(L) (*function*), 67
MaximalIntegralLattice(L, p) (*function*), 67
MaximalIntegralLattice(Q) (*function*), 67
MaximalIntegralLattice(V) (*function*), 67

MaximalNormSplitting(L, p) (*function*), 70
MaximalSublattices(e) (*function*), 103
MaximalSublattices(L, p) (*function*), 66
Min(L) (*function*), 31
MinimalSuperlattices(e) (*function*), 103
MinimalSuperlattices(L, p) (*function*), 66
Minimum(L) (*function*), 31
MinkowskiGramReduction(M) (*function*), 29
MinkowskiLattice(I) (*function*), 7
MinkowskiLattice(O) (*function*), 7
MinkowskiReduction(L) (*function*), 29
MinkowskiSpace(K) (*function*), 7
Module(L) (*function*), 4, 65
MooreDeterminant(M) (*function*), 59
Morphism(e) (*function*), 103

N

Nagens(L) (*function*), 96
Names in the Lattice Database (*example*), 57
NaturalActionGenerator(L, i) (*function*), 96
NaturalGroup(L) (*function*), 96
Neighbor(L, v, p) (*function*), 52
Neighbor(nProc) (*function*), 75
NeighborClosure(L, p) (*function*), 53
NeighborProcess(L, P, k) (*function*), 75
Neighbors(L, p) (*function*), 53
Neighbour(L, v, p) (*function*), 52
NeighbourClosure(L, p) (*function*), 53
Neighbours(L, p) (*function*), 53, 72
NextVector(P) (*function*), 40
Nontrivial Coefficient Ideals (*example*), 84
Norm(L) (*function*), 69
Norm(v) (*function*), 10, 79
NumberFieldLattice(D) (*function*), 64
NumberFieldLattice(K, d) (*function*), 63
NumberFieldLattice(L) (*function*), 64
NumberFieldLattice(S) (*function*), 63
NumberFieldLatticeWithGram(F) (*function*), 64
NumberOfActionGenerators(L) (*function*), 96
NumberOfAntisymmetricForms(L) (*function*), 97
NumberOfInvariantForms(L) (*function*), 97
NumberOfIsotropicSubspaces(L, P, k) (*function*), 74
NumberOfLattices(D) (*function*), 57
NumberOfLattices(D, d) (*function*), 57
NumberOfLattices(D, N) (*function*), 57
NumberOfLevels(V) (*function*), 102
NumberOfNeighbors(L, P, k) (*function*), 75
NumberOfSymmetricForms(L) (*function*), 97
NumericalSignature(M) (*function*), 49

O

Oddity(f) (*function*), 108
Oddity(G) (*function*), 52
Oddity(L) (*function*), 108
Oddity(M) (*function*), 108
Operations on Lattice Elements (*example*), 11
Orbit(G, v) (*function*), 79
Order(f) (*function*), 115
Order(L) (*function*), 14, 69
OrthogonalComplement(L, S) (*function*), 66
OrthogonalComplement(L, v) (*function*), 66
OrthogonalDecomposition(F) (*function*), 18
OrthogonalDecomposition(L) (*function*), 18
OrthogonalDecomposition(L, F) (*function*), 18
Orthogonalization(F) (*function*), 47
Orthogonalize(L) (*function*), 48
Orthogonalize(M) (*function*), 47
OrthogonalizeGram(F) (*function*), 47
OrthogonalizeGram(F, a) (*function*), 47
Orthogonalizing the Dual Coxeter-Todd Lattice (*example*), 48
OrthogonalSum(L, M) (*function*), 17
Orthonormalize(L) (*function*), 48
Orthonormalize(L, K) (*function*), 48
Orthonormalize(M) (*function*), 48
Orthonormalize(M, K) (*function*), 48

P

PackingRadius(L) (*function*), 31
pAdicDiagonalization(L, p) (*function*), 50
pAdicJordanDecomposition(L, p) (*function*), 50
PairReduce(L) (*function*), 26
PairReduce(X) (*function*), 25
PairReduceGram(F) (*function*), 26
Parallel Enumeration (*example*), 44
Parent(f) (*function*), 116
Parent(v) (*function*), 79
PartialDual(L, n) (*function*), 16
pExcess(f, p) (*function*), 108
pExcess(L, p) (*function*), 108
pExcess(M, p) (*function*), 108
PositiveDefiniteForm(L) (*function*), 97
PositiveDefiniteQuadraticLattices(n, d) (*function*), 53
Power(f, n) (*function*), 115
Prime(G) (*function*), 52
PrimeForm(Q, p) (*function*), 114
Primes(G) (*function*), 51
Primes(V) (*function*), 103
ProperAutomorphismGroup(L) (*function*), 77, 89
PseudoBasis(L) (*function*), 13, 68
PseudoBasisMatrix(L) (*function*), 68
PseudoGramMatrix(L) (*function*), 69
PseudoMatrix(L) (*function*), 68
pSignature(f,p) (*function*), 107
pSignature(L,p) (*function*), 107
pSignature(M,p) (*function*), 107
PureLattice(L) (*function*), 18

Q

Q ! 1 (*operation*), 113
Q ! f (*operation*), 116
Q ! [a, b, c] (*operation*), 114
QuadraticForm(L) (*function*), 13, 107
QuadraticForm(M) (*function*), 107

QuadraticFormInvariants(f) (*function*), 109
QuadraticFormInvariants(L) (*function*), 109
QuadraticFormInvariants(M) (*function*), 109
QuadraticForms(D) (*function*), 113
QuadraticFormWithInvariants(n, d, F, N) (*function*), 107
QuadraticLattice(n, d) (*function*), 4
QuadraticOrder(Q) (*function*), 116
QuaternaryLatticeOfPrimeDiscriminant(p) (*function*), 4
QuaternaryQuadraticLattices(d) (*function*), 53
Quaternionic Automorphism Group of the Leech Lattice (*example*), 60
QuaternionicAutomorphismGroup(M) (*function*), 59
QuaternionicBasis(M) (*function*), 59
QuaternionicGModule(M, I, J) (*function*), 59
QuaternionicTranspose(M) (*function*), 59
QuinaryLatticeOfPrimeDiscriminant(p) (*function*), 4
QuinaryQuadraticLattices(d) (*function*), 53
quo< L | S > (*constructor*), 15
QuotientMap(Q1, Q2) (*function*), 116

R

r * L (*operation*), 65
RandomLattice(n, M) (*function*), 4
Rank(G) (*function*), 51, 52
Rank(L) (*function*), 12, 69
RationalRepresentative(G) (*function*), 51
Recommended Uses of LLL (*example*), 20
ReconstructLatticeBasis(S, B) (*function*), 30
ReducedForm(f) (*function*), 115
ReducedForms(D) (*function*), 117
ReducedForms(Q) (*function*), 117
ReducedOrbits(Q) (*function*), 117
Reduction(f) (*function*), 115
ReductionOrbit(f) (*function*), 115
ReductionStep(f) (*function*), 115
RepresentationNumber(f, n) (*function*), 119
Representative(G) (*function*), 51, 52
Representatives(G) (*function*), 53
RestrictionOfScalarsToZ(L) (*function*), 66

S

s * L (*operation*), 15
s * v (*operation*), 9, 79
S subset L (*operation*), 76
Scale(L) (*function*), 13, 70
ScaledLattice(L, n) (*function*), 4
ScaledLattice(L, r) (*function*), 65
SetVerbose("Enum", v) (*procedure*), 42
SetVerbose("HKZ", v) (*procedure*), 28
SetVerbose("LLL", v) (*procedure*), 24
Seysen Reduction of the Leech Lattice (*example*), 27
Seysen(L) (*function*), 27
Seysen(X) (*function*), 26
SeysenGram(F) (*function*), 26
Shortest Vectors of the E8 Lattice (*example*), 34
ShortestVector(L) (*function*), 33
ShortestVectors(G) (*function*), 95
ShortestVectors(L) (*function*), 33
ShortestVectorsMatrix(L) (*function*), 33
ShortVector(L, l, u) (*function*), 35
ShortVector(L, u) (*function*), 35
ShortVectors(G, B) (*function*), 95
ShortVectors(L, l, u) (*function*), 35
ShortVectors(L, u) (*function*), 35
ShortVectorsMatrix(L, l, u) (*function*), 35
ShortVectorsMatrix(L, u) (*function*), 35
ShortVectorsProcess(L, l, u) (*function*), 40
ShortVectorsProcess(L, u) (*function*), 40
Signature(F) (*function*), 49, 75
Signature(G) (*function*), 51
SimpleLattice(L) (*function*), 66
SkipTo(nProc, V, S) (*procedure*), 75
Solving a Knapsack Problem (*example*), 36
Sphere(L, e) (*function*), 78
SpinorCharacters(G) (*function*), 51
SpinorGenera(G) (*function*), 49
SpinorGenerators(G) (*function*), 51
SpinorGenus(L) (*function*), 49
SpinorNorm(L, p) (*function*), 72
SpinorRepresentatives(L) (*function*), 53
Splitting Representations with Short Vectors (*example*), 38
Stabilizer(G, v) (*function*), 79
StandardLattice(n) (*function*), 4
StandardLattice(V) (*function*), 64
sub<L | RHS> (*constructor*), 65
sub<L | S> (*constructor*), 14
SublatticeClasses(G) (*function*), 99
SublatticeLattice(G) (*function*), 102
SublatticeLattice(G, p) (*function*), 102
SublatticeLattice(G, Q) (*function*), 102
Sublattices of a G-Lattice (*example*), 99
Sublattices with Projections (*example*), 101
Sublattices(G) (*function*), 98
Sublattices(G, p) (*function*), 98
Sublattices(G, Q) (*function*), 98
Sublattices(L) (*function*), 98
Sublattices(L, p) (*function*), 98
Sublattices(L, Q) (*function*), 98
Sublattices, Superlattices, and Quotients (*example*), 16
Successive Minima of a Lattice (*example*), 41
SuccessiveMinima(L) (*function*), 41
SuccessiveMinima(L, k) (*function*), 41
Support(v) (*function*), 10
SymmetricForms(L) (*function*), 97
SymmetricForms(L, n) (*function*), 97
SymmetricMatrix(f) (*function*), 107
SymmetricSquare(L) (*function*), 18

T

T * L (*operation*), 15, 66
T * v (*operation*), 79
TensorProduct(L, M) (*function*), 18
TernaryQuadraticLattice(d) (*function*), 4
TernaryQuadraticLattices(d) (*function*), 53
The Maass Unimodular Lattice (*example*), 85
ThetaSeries(f, n) (*function*), 119
ThetaSeries(L, n) (*function*), 41
ThetaSeriesIntegral(L, n) (*function*), 42
ThetaSeriesModularForm(L) (*function*), 45
ThetaSeriesModularFormSpace(L) (*function*), 45
TJ * L (*operation*), 66
Trains(G) (*function*), 52
TwoTorsionSubgroup(Q) (*function*), 118
Type(L) (*function*), 12

V

V ! i (*operation*), 102
V ! M (*operation*), 102
v * s (*operation*), 9, 79
v * T (*operation*), 10, 79
v *:= n (*operation*), 10
v + w (*operation*), 9, 79
v +:= w (*operation*), 9
v / s (*operation*), 9, 79
v - w (*operation*), 9, 79
v -:= w (*operation*), 10
v ^ G (*operation*), 79
v ^ M (*operation*), 79
v div d (*operation*), 9
v eq w (*operation*), 10, 79
v in L (*operation*), 10, 78
v ne w (*operation*), 10, 79
Vector(v) (*function*), 80
Volume(L) (*function*), 69
Voronoi Cell of a Perfect Lattice (*example*), 46
VoronoiCell(L) (*function*), 45
VoronoiGraph(L) (*function*), 46
VoronoiRelevantVectors(L) (*function*), 46

W

WeierstrassSeries(z, f) (*function*), 120
Witt Index of a Random Form (*example*), 110
WittInvariant(f, p) (*function*), 108
WittInvariant(L, P) (*function*), 108
WittInvariant(L, p) (*function*), 71, 108
WittInvariant(M, p) (*function*), 71, 108
WittInvariants(f) (*function*), 109
WittInvariants(L) (*function*), 109
WittInvariants(M) (*function*), 109

Z

Zero(L) (*function*), 9, 78
ZLattice(L) (*function*), 4, 66