

Exploring Lattices and Algebraic Structures in SageMath

CyBOK © Crown Copyright, The National Cyber Security Centre 2025, licensed under the Open Government Licence
<http://www.nationalarchives.gov.uk/doc/open-government-licence/>

Introduction

This SageMath sheet is intended to provide a practical guide for working with lattice-based computations. It covers fundamental operations, including vector and matrix manipulations, polynomial arithmetic, random element generation, and various lattice-related tasks. The goal is to offer a hands-on approach to exploring lattice-based cryptography and other lattice-related mathematical concepts, utilising SageMath's powerful computational tools.

It assumes that the reader has basic familiarity with SageMath and its syntax.

SageMath is a free, open-source, and powerful mathematics software system licensed under the GPL. It combines many existing open-source packages into a common Python-based interface. For the remainder of this document, we will refer to it simply as Sage.

You can work with Sage in different ways:

- **Sage Cell:** Run Sage code online instantly via <https://sagecell.sagemath.org/>.
- **CoCalc:** Use an online collaborative platform at <https://cocalc.com/>, which provides Sage alongside Jupyter notebooks and other tools.
- **Local Installation:** Install Sage on your computer by downloading it from <https://www.sagemath.org/>, allowing for offline use and better performance.

For more information about Sage, please refer to <https://doc.sagemath.org/html/en/tutorial/index.html>.

CyBOK Mapping

The material in this sheet maps to the following knowledge areas of the CyBOK:

- **Systems Security** → **Cryptography**
- **Infrastructure Security** → **Applied Cryptography**

Contents

1	Prime Numbers	2
2	$\mathbb{Z}_n$, $\mathbb{Z}_n^\times$, and Finite Fields	3
3	Polynomial Rings	4
3.1	Quotient Polynomial Rings	5

4	Vectors	5
4.1	Defining Vectors & Basic Vector Operations	5
4.1.1	Vectors Over Rings and Finite Fields	6
4.2	Vector Norm	6
4.3	Random Vectors	6
5	Matrices	7
5.1	Matrix Operations	7
5.2	Matrix Transpose	8
5.3	Matrix Determinant	8
5.4	Matrix Inversion	8
5.5	Random Matrices	9
6	Sampling Distributions	9
6.1	Discrete Gaussian Sampling Over the Integers	9
6.2	Discrete Gaussian over Lattices	10
7	Lattices	10
7.1	Creating Lattice using <code>IntegerLattice</code>	11
7.2	Creating Lattices using <code>gen_lattice</code>	12
7.2.1	Creating a Primal Lattice	12
7.2.2	Creating a Dual Lattice	12
7.2.3	Creating an Ideal Lattice	12
7.2.4	Additional Examples of Using <code>gen_lattice</code>	13
7.3	Lattice Reductions	13
7.3.1	Using the LLL Algorithm	13
7.3.2	Using the BKZ Algorithm	14
8	LWE and RingLWE Modules	14

1 Prime Numbers

Sage provides several functions for working with prime numbers.

Random Primes

The function `random_prime(n)` generates a random prime number $\leq n$.

```
# Returns a random prime <= 1000
p = random_prime(1000)
print(p)
```

We can also specify a lower bound using the `lbound` parameter:

```
# Random prime from the integer range [500, 1000]
p = random_prime(1000, lbound=500)
print(p)
```

Checking for Primality

To check whether an integer x is prime, use the function `is_prime()`.

```
x = 13
# Check if x is a prime
x.is_prime()
```

Finding the Next or Previous Prime

To find the smallest prime number greater than x :

```
x.next_prime()
```

To find the largest prime number smaller than x :

```
x.previous_prime()
```

2 $\mathbb{Z}_n$, $\mathbb{Z}_n^\times$, and Finite Fields

The Ring of Integers Modulo n and the Multiplicative Group $\mathbb{Z}_n^\times$

The ring $\mathbb{Z}_n$, which consists of the set $\{0, 1, 2, \dots, n-1\}$ with modular addition and multiplication, can be defined in Sage using the command `IntegerModRing(n)`. Below is an example that lists the elements of $\mathbb{Z}_{10}$:

```
Z10 = IntegerModRing(10)
# Perform 5*3 in this ring
Z10(5) * Z10(3)
Z10.list()
```

The set $\mathbb{Z}_n^\times$ consists of the elements of $\mathbb{Z}_n$ that are coprime to n , i.e., the units of the ring. The following example lists the elements of $\mathbb{Z}_{10}^\times$:

```
# Define the ring Z_{10} (integers modulo 10)
# Alternative method to do this is: Z10 = Zmod(10)
Z10 = IntegerModRing(10)

# List all elements in the multiplicative group Z_{10}^*
Z10.list_of_elements_of_multiplicative_group()

# The group of multiplicative units (invertible elements)
Z10S = Z10.unit_group()
```

Finite Fields

A finite field (or Galois field) is a field with a finite number of elements. The simplest example is $\mathbb{Z}_p$ where p is a prime number, which forms a field under modular arithmetic.

We can define a finite field using `GF(q)`, where q is a prime power. For example, $\mathbb{F}_{13}$ (the finite field with 13 elements) can be defined as follows:

```
F13 = GF(13)
```

Elements of the field can be used for arithmetic operations just like integers:

```
F13(15) * F13(11)
```

Finding the Multiplicative Group

The multiplicative group of a finite field consists of all nonzero elements. The following command lists its elements:

```
F13.multiplicative_group().list()
```

This returns the elements of $\mathbb{F}_{13}^*$, i.e., all nonzero elements of the field.

Finite Fields of Prime Power Order

If the field size is a prime power p^k , Sage requires specifying an indeterminate (either explicitly or implicitly) when constructing the field. The following example defines $\mathbb{F}_{2^3}$, the finite field with $2^3 = 8$ elements:

```
F8.<x> = GF(2^3)
```

Elements of a finite field can be added, multiplied, and inverted:

```
# Get 2 random elements from F8
a = F8.random_element()
b = F8.random_element()

c = a + b
d = a * b

# Multiplicative inverse of a
inv_a = a^(-1)
```

3 Polynomial Rings

Sage has functions and tools to simplify working with polynomials.

In the below example we show how to define the ring of polynomials over the integer ring $\mathbb{Z}_2$, i.e. $\mathbb{Z}_2[X]$. We explore several properties of a given polynomial. The operations include checking irreducibility, factorisation, primitivity, and divisibility in the ring $\mathbb{Z}_2[X]$. Note $\mathbb{Z}_2$ itself is field since 2 is a prime.

```
# The ring Z_2 (integers mod 2)
Z2 = IntegerModRing(2)

# The polynomial ring Z_2[X]
R.<X> = PolynomialRing(Z2)

# c is the polynomial X^2 + X in Z_2[X]
c = X^2 + X

# Check whether the polynomial is irreducible
c.is_irreducible()

# Factor the polynomial
c.factor()

# Check whether the polynomial is primitive
c.is_primitive()

# Check whether the polynomial c divides the polynomial X^4 + 1
c.divides(X^4 + 1)

# Sample a random polynomial of degree 3 from Z_2[X]
print(R.random_element(degree=3))
```

3.1 Quotient Polynomial Rings

A quotient polynomial ring is constructed by taking a polynomial ring over a base ring and factoring out an ideal generated by a polynomial. This quotient ring is particularly useful in lattice-based cryptography, where operations are performed modulo a cyclotomic polynomial such as $X^n + 1$.

The below example shows how to create the ring $\mathbb{Z}[X]/(X^{11} + 1)$. This type of ring is important in lattice-based cryptography, particularly in schemes such as those relying on Ring-LWE.

```
# Define polynomial ring over integers
R.<X> = PolynomialRing(ZZ)

# Modulus polynomial
f = X^11 + 1

# Define the quotient ring
Q.<X> = R.quotient(f)

# Example to multiply elements in the quotient ring
print((X^2 + 3) * (X^5 + 1))

# Sampling a random element from the quotient ring
print(Q.random_element())
```

Here, the ring Q represents $\mathbb{Z}[X]/(X^{11} + 1)$, where arithmetic operations are performed modulo $X^{11} + 1$.

4 Vectors

Sage provides built-in support for vectors, allowing easy operations such as addition, dot products, and norms.

4.1 Defining Vectors & Basic Vector Operations

Below are some examples of how to define a vector.

```
# Define a vector v with (integer) components (1, 2, 3)
v = vector([1, 2, 3])

# Define another vector w with (integer) components (4, 5, 6)
w = vector([4, 5, 6])
```

Vectors support addition and subtraction component-wise.

```
# Perform vector addition: (1,2,3) + (4,5,6) = (5,7,9)
v + w

# Perform vector subtraction: (1,2,3) - (4,5,6) = (-3,-3,-3)
v - w
```

The dot product of two vectors can be computed using either the `*` operation or by using the `dot_product()` function.

```
# Compute dot product: (1*4 + 2*5 + 3*6) = 32
v * w

# Compute dot product: (1*4 + 2*5 + 3*6) = 32
v.dot_product(w)
```

4.1.1 Vectors Over Rings and Finite Fields

The default for vectors is integer coordinates. Sage allows defining vectors over various mathematical structures, including integers, finite fields, and polynomial rings.

Vectors can be defined over a finite field $\mathbb{F}_q$.

```
# Define the finite field GF(7)
F = GF(7)

# Vector v over GF(7) (its entries are elements of GF(7))
v = vector(F, [3, 4, 5])
```

Vectors can also have entries from a polynomial ring.

```
# Polynomial ring over the integers in the indeterminate X (i.e. ZZ[X])
R.<X> = ZZ[]

# Vector with polynomial entries
v = vector(R, [X, X^2 + 1, X^2 - X])
```

One can find the parent of an object (e.g. a vector) by using the `.parent()` method. The parent refers to the mathematical structure/set to which the object belongs.

```
# Prints the name of the parent structure of v
print(v.parent())
```

4.2 Vector Norm

The p th norm (length $\|\cdot\|_p$) of a vector can be computed by calling the `norm(p)` function, where p specifies the norm to compute. For example, setting $p = 2$ yields the Euclidean norm, $p = 1$ gives the Manhattan norm, and $p = \text{infinity}$ produces the L_∞ norm. If no parameter is provided, the Euclidean norm is used by default.

```
# Define a vector v with (integer) components (1, 2, 3)
v = vector([1, 2, 3])

# Compute the Euclidean norm ||v|| = sqrt(1^2 + 2^2 + 3^2) = sqrt(14)
v.norm(2)

# Computes L_infinity-norm = max(|1|, |2|, |3|) = 3
v.norm(infinity)
```

4.3 Random Vectors

One can generate random vectors using the function `random_vector`. In the example below, the last two parameters of `random_vector` (which are optional) can be used to specify the range from which the random values are drawn. Here, x represents the minimum value and y represents the maximum value.

```
# v is a randomly generated vector with 3 coordinates and values drawn from
# the ring of integers.
# The random values are drawn from the range [0,19].
v = random_vector(ZZ, 3, x=0, y=19)
```

5 Matrices

We can define a matrix using the `Matrix()` data structure. Some examples are below.

```
# Define a 2x2 matrix (with integer values)
A = Matrix([[1, 2], [3, 4]])

# Show the matrix
A
```

This defines the matrix:

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

You can create matrices over a variety of rings and structures, such as finite fields, integer rings, and polynomial rings. Below is an example of creating a matrix over a polynomial ring.

```
# Define a polynomial ring over the integers (i.e. Z[X])
R.<X> = ZZ[]

# Define a 2x2 matrix with polynomial entries
M = Matrix(R, [[X+1, X^2], [X^3, X^4]])

# Display the matrix
print(M)
```

In this example, the matrix $\mathbf{M}$ is a 2×2 matrix where the entries are polynomials in X with integer coefficients. You can perform matrix operations such as addition, multiplication, and finding the determinant, all of which will be computed in the context of the polynomial ring.

5.1 Matrix Operations

Addition

Matrix addition is performed element-wise.

```
# Define a 2x2 matrix A
A = Matrix([[1, 2], [3, 4]])

# Define a 2x2 matrix B
B = Matrix([[5, 6], [7, 8]])

# Matrix addition
C = A + B
```

$$\mathbf{C} = \mathbf{A} + \mathbf{B} = \begin{bmatrix} 1+5 & 2+6 \\ 3+7 & 4+8 \end{bmatrix} = \begin{bmatrix} 6 & 8 \\ 10 & 12 \end{bmatrix}$$

Multiplication

We can multiply matrices as long as they are compatible: the number of columns of the first must be equal to the number of rows of the second.

```
# Matrix multiplication
C = A * B
```

$$\mathbf{C} = \mathbf{A} \times \mathbf{B} = \begin{bmatrix} (1 \cdot 5 + 2 \cdot 7) & (1 \cdot 6 + 2 \cdot 8) \\ (3 \cdot 5 + 4 \cdot 7) & (3 \cdot 6 + 4 \cdot 8) \end{bmatrix} = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

Matrix-Vector Multiplication

Vectors can be multiplied by matrices when dimensions align.

```
# Define a 2-dimensional vector
v = vector([5, 6])

# Define a 2x2 matrix M
A = Matrix([[1, 2], [3, 4]])

# Multiply matrix M by vector v (as a column vector)
M * v
```

$$\mathbf{M} \times \mathbf{v} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 5 \\ 6 \end{bmatrix} = \begin{bmatrix} 17 \\ 39 \end{bmatrix}$$

5.2 Matrix Transpose

The transpose of a matrix swaps its rows and columns.

```
# Define a 2x2 matrix M
M = Matrix([[1, 2], [3, 4]])

# Compute matrix M's transpose
M.transpose()
```

This should print:

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

5.3 Matrix Determinant

The determinant of a matrix is computed using the `determinant()` function.

```
# Define a 2x2 matrix M
M = Matrix([[1, 2], [3, 4]])

# Compute determinant of matrix M
M.determinant()
```

$$\det(\mathbf{M}) = (1 \cdot 4) - (2 \cdot 3) = -2$$

5.4 Matrix Inversion

If a matrix is invertible (its determinant $\neq 0$), we can compute its inverse.

```
# Define a 2x2 matrix M
M = Matrix([[1, 2], [3, 4]])

# Compute matrix M's inverse
M.inverse()
```

$$\mathbf{M}^{-1} = \begin{bmatrix} -2 & 1 \\ \frac{3}{2} & -\frac{1}{2} \end{bmatrix}$$

5.5 Random Matrices

You can generate a random matrix over a specified ring/field using the function `random_matrix`. Below are some examples.

```
# Create a random 2 x 1 matrix over GF(2) (field of size 2)
A = random_matrix(GF(2), 2, 1)
print(A)

# Generate a random 3 x 3 matrix over integers, with entries in the range
  [-10, 10]
B = random_matrix(ZZ, 3, 3, x=-10, y=10)
print(B)
```

In this example the first matrix, **A**, is a random 2×1 matrix over $\mathbb{Z}_2$, where each entry is randomly chosen from $\{0, 1\}$. The second matrix, **B**, is a random 3×3 matrix with integer entries, each between -10 and 10 .

6 Sampling Distributions

A key component of many lattice-based cryptosystems is the concept of sampling from specific distributions, which introduces noise to ensure security, such as in LWE-based systems.

6.1 Discrete Gaussian Sampling Over the Integers

The *Discrete Gaussian Distribution* (centred around 0) is a probability distribution defined over integers, where the probability of each integer is weighted by a Gaussian function. Mathematically, it is defined as:

$$P(x) = \frac{e^{-\frac{x^2}{2\sigma^2}}}{Z(\sigma)}$$

where:

- $x \in \mathbb{Z}$ is an integer.
- σ is the standard deviation, which controls how spread out the distribution is from the center.
- $Z(\sigma)$ is the normalization factor to ensure the total probability sums to 1.

Sage provides functions for sampling from discrete Gaussian distributions. Below is an example of how to generate samples from a discrete Gaussian distribution with standard deviation 2.5 and mean 0.

```
# Import the module
from sage.stats.distributions.discrete_gaussian_integer import
  DiscreteGaussianDistributionIntegerSampler

# Define the distribution with sigma=2.5 and mean 0 (centred around 0)
D = DiscreteGaussianDistributionIntegerSampler(2.5)

# Sample 2^20 samples from the distribution and draw a histogram
histogram([D() for _ in range(2^20)], color="grey")
```

Figure 1 shows visualisation of the histogram for 2^{20} samples.

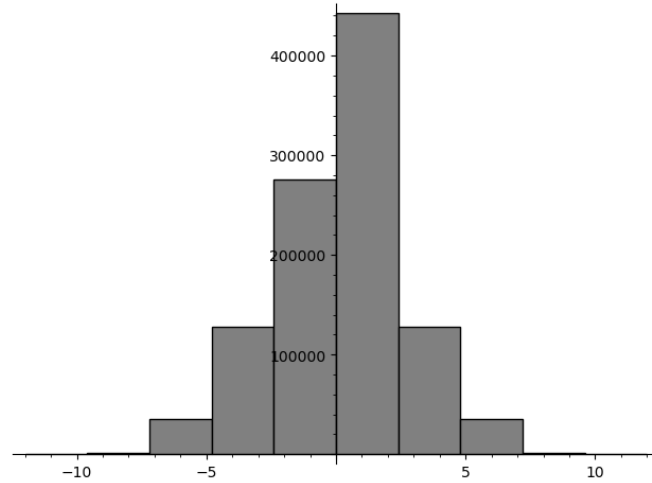


Figure 1: Visualization of a discrete Gaussian histogram with $\sigma = 2.5$ for 2^{20} samples.

6.2 Discrete Gaussian over Lattices

Sage also provides discrete Gaussian distribution functionalities over lattices, allowing us to sample lattice vectors. We can use the class `DiscreteGaussianDistributionLatticeSampler` [6].

In the example below, we demonstrate how to use the sampling algorithm from [1] to sample a 2-dimensional lattice over $\mathbb{Z}$, which is generated by the 2×2 identity matrix. That is, the basis of the lattice is $\mathbf{b}_1 = (1, 0)$ and $\mathbf{b}_2 = (0, 1)$, where the distribution has a standard deviation of 3.0.

Figure 2 shows a plot for 2^{20} samples, illustrating the distribution of sampled lattice points.

```
# Import the module
from sage.stats.distributions.discrete_gaussian_lattice import
    DiscreteGaussianDistributionLatticeSampler

# Define the distribution with sigma=3 over the lattice generated by basis
    (1,0) and (0,1)
D = DiscreteGaussianDistributionLatticeSampler(identity_matrix(2), 3.0)

# Draw 2^20 samples from D
sampleList = [D() for _ in range(2^20)]

# Prepare for plotting
vcountList = [vector(x.list() + [sampleList.count(x)]) for x in set(sampleList
)]

# Plot the graph
list_plot3d(vcountList, point_list=True, interpolation="nn")
```

7 Lattices

In this section we explain some methods that you can use to create lattices.

We now proceed to showing some example ways of defining lattices in Sage.

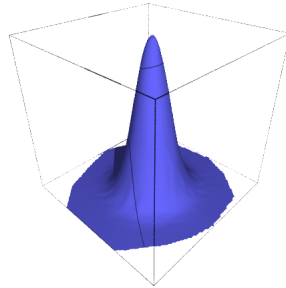


Figure 2: Visualization plot for 2^{20} samples from the lattice.

7.1 Creating Lattice using IntegerLattice

To create an integer Lattice from a basis matrix, you can use the `IntegerLattice` module [7] which needs to be imported from `sage.modules.free_module_integer`. Below is an example of how to use this.

```
# Import the module
from sage.modules.free_module_integer import IntegerLattice

# Define a matrix (basis) for the lattice
B = Matrix([[1, 2], [3, 5]])

# Define the lattice L(B)
L = IntegerLattice(B)

# Display L(B)
L

# Display the basis matrix of the lattice
L.basis_matrix()
```

The above example creates a lattice using the matrix **B** as its basis. In the above code, we used a 2×2 matrix for the basis as an example. Note that the basis you will get in the above example will be:

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

which is different from our matrix **B**. The reason is that Sage, by default, reduces the basis. To force Sage not to reduce the basis, you can use: `L = IntegerLattice(B, lll_reduce=False)` to prevent Sage from reducing the basis.

Below is another example for creating a lattice whose basis is 50×50 random matrix over the integers.

```
# Import the module
from sage.modules.free_module_integer import IntegerLattice

# Generate a random 50x50 matrix with integer entries between -500 and 500
B = random_matrix(ZZ, 50, 50, x=-500, y=500)

# Create an integer lattice from the matrix B
L = IntegerLattice(B)

# Compute the norm of the shortest vector in the lattice
result = L.shortest_vector().norm().log(2).n()
```

7.2 Creating Lattices using `gen_lattice`

Alternatively, one can use `sage.crypto.lattice.gen_lattice` [5], which by default returns a random matrix basis for the lattice rather than a lattice object. If one wishes to obtain a lattice object instead of just a basis matrix, one can add the parameter `lattice=True`.

We show below some examples of how to use this method.

7.2.1 Creating a Primal Lattice

The below example shows how to create a lattice object (rather than just a basis matrix) for the Primal q -ary lattice $L(\mathbf{A})$ for $\mathbf{A} \in \mathbb{Z}_q^{m \times n}$ where

$$L(\mathbf{A}) = \{\mathbf{x} \in \mathbb{Z}^m \mid \exists \mathbf{y} \in \mathbb{Z}_q^n, \mathbf{x} \equiv \mathbf{A}\mathbf{y} \bmod q\}$$

The parameter m specifies the dimension of the lattice, while n specifies the volume of the lattice (its determinant), which will be an integer (i.e., the volume is q^n).

```
# Import the module
from sage.crypto.lattice import gen_lattice

# Create the random lattice over Z^5
L = gen_lattice(m=5, n=3, q=17, seed=41, dual=False, lattice=True)

# Print the lattice volume
print(L.volume())

# Choose a random element from the lattice
print(L.random_element())

# Check if the vector (1,2,3,4,5) is in the lattice
print(vector([1,2,3,4,5]) in L)
```

7.2.2 Creating a Dual Lattice

The below example shows how to create the dual q -ary lattice $L^\perp(\mathbf{A}^T)$ for $\mathbf{A} \in \mathbb{Z}_q^{m \times n}$. This is defined as:

$$L^\perp(\mathbf{A}^T) = \{\mathbf{x} \in \mathbb{Z}^m \mid \mathbf{A}^T \mathbf{x} \equiv \mathbf{0} \bmod q\}$$

The parameter m specifies the dimension of the lattice, while n specifies the volume of the lattice (its determinant), which will be an integer (i.e., the volume is q^{m-n}).

```
# Import the module
from sage.crypto.lattice import gen_lattice

# Create the random dual lattice over Z^5
LDual = gen_lattice(m=5, n=3, q=17, seed=41, dual=True, lattice=True)

# Print the lattice volume
print(LDual.volume())

# Choose a random element from the lattice
print(LDual.random_element())

# Check if the vector (1,2,3,4,5) is in the lattice
print(vector([1,2,3,4,5]) in LDual)
```

7.2.3 Creating an Ideal Lattice

`sage.crypto.lattice.gen_lattice` can also be used to create ideal lattices, which are ideals in the polynomial ring $\mathbb{Z}[x]/(f(x))$, such as those used for Ring-SIS and Ring-LWE. Below is an example where the quotient polynomial is $f(x) = x^4 - 1$. Note that when using this mode, if the parameter n is not omitted, it must match the degree of $f(x)$.

```
# Import the module
from sage.crypto.lattice import gen_lattice

# Define the polynomial ring ZZ[x]
R.<x> = PolynomialRing(ZZ)

# Generate an ideal lattice where f(x) = x^4 - 1
LX = gen_lattice(type="ideal", seed=27, quotient=x^4 - 1, lattice=True)

# Print the lattice volume
print(LX.volume())

# Choose a random element from the lattice
print(LX.random_element())
```

7.2.4 Additional Examples of Using gen_lattice

Below are two examples demonstrating other uses of the `sage.crypto.lattice.gen_lattice`. In these examples, the parameters are interpreted as follows:

- m is the dimension of the lattice (i.e., the lattice is a subset of $\mathbb{Z}^m$).
- n specifies the exponent in the volume (determinant) of the lattice, so that the volume is q^n .
- q is the modulus (coefficient size).

```
# Import the module
from sage.crypto.lattice import gen_lattice

# Creating a random 4D Lattice with determinant (volume) (determined by q^n)
, and a random seed 36 is used
L1 = gen_lattice(type="random", n=1, m=4, q=1013, seed=36, lattice=True)

# Print Volume
print(L1.volume())

# Creating a random 5D modular dual lattice with determinant (volume) (
determined by q^{m-n}) and a random seed 39 is used
L2 = gen_lattice(type="modular", n=2, m=5, q=1013, seed=39, dual=True, lattice=
True)

# Print Volume
print(L2.volume())
```

7.3 Lattice Reductions

Sage has implementations of both the Lenstra-Lenstra-Lovász (LLL) [2] and Block Korkine-Zolotarev (BKZ) [4] algorithms, which can be used to reduce matrices and lattice bases to shorter and more orthogonal forms.

7.3.1 Using the LLL Algorithm

Consider the following 3x3 matrix:

$$\mathbf{M} = \begin{bmatrix} 10 & -50 & 100 \\ 37 & 18 & 20 \\ 13 & 105 & 26 \end{bmatrix}$$

In the below example we show how to use LLL with $\delta = 0.65$ to reduce the matrix. The optional parameter δ where $0.25 < \delta < 1$ controls how strict the reduction is and it is a trade-off between quality of reduction and efficiency. Note that parameters are optional.

```
# Create a 3x3 matrix
M = matrix([[10, -50, 100], [37, 18, 20], [13, 105, 26]])

# Reduce M using LLL
print (M.LLL(delta = 0.65))
```

The above example will print the below reduced matrix.

$$\begin{bmatrix} 37 & 18 & 20 \\ -27 & -68 & 80 \\ -24 & 87 & 66 \end{bmatrix}$$

7.3.2 Using the BKZ Algorithm

The BKZ algorithm is another method for reducing lattice bases. For exact parameters of the algorithm, we refer the reader to the Sage manual [8].

Below is an example of how you can call the BKZ algorithm.

```
# Create a random 20x20 matrix with entries from the integer range [-100,200]
M = random_matrix(ZZ, 20, 20, x=-100, y=200)

# Reduce M using BKZ
print (M.BKZ(block_size=12))
```

The parameter `block_size` in the above example specifies the number of blocks in the reduction, where a higher value will yield shorter vectors but at the expense of running time.

8 LWE and RingLWE Modules

The Sage `LWE` and `RingLWE` modules [9] provide tools for working with the Learning With Errors (LWE) problem and its ring variant R-LWE, which are foundational problems in lattice-based cryptography. LWE is widely used in creating cryptographic constructions. The module provides various LWE and R-LWE instances, including those for a number of existing settings in the literature, e.g., [3]. Using these tools, one can generate LWE and R-LWE instances with parameters of their choice.

It is worth noting that in Sage, arithmetic in modular rings $(\mathbb{Z}_q)$ is in the range $[0, q - 1]$. However, for many lattice-based cryptographic applications, it is beneficial to use a *centred* representation in the range $[-\frac{q}{2}, \frac{q}{2}]$. This helps, for example, reduce noise growth, ensure correct decryption, and maintain symmetry in error distributions.

To convert an element $x \in [0, q - 1]$ to its *centred* equivalent x_{centred} , one should use the following transformation:

$$x_{\text{centred}} = \begin{cases} x, & \text{if } x \leq \frac{q}{2} \\ x - q, & \text{if } x > \frac{q}{2} \end{cases}$$

For instance, in $\mathbb{Z}_7$ in Sage, -2 corresponds to 5 , whereas its *centred* version is -2 . Similarly, 4 in $\mathbb{Z}_7$ is 4 , whereas its *centred* version is -3 .

One can use the `sage.crypto.lwe.balance_sample` function when working with LWE (or Ring-LWE) samples, which takes two parameters: the first being the sample that needs centring and q (which is optional). If q is not provided, the sample is assumed to live in the integers. If no q is given, the sample's elements (the vector $\mathbf{a}$ and scalar b) are assumed to belong to $\mathbb{Z}_q$.

Below we provide some examples of how to use the `LWE` module. One can also experiment with the `RingLWE` module.

```
# Import the Discrete Gaussian Distribution module
from sage.stats.distributions.discrete_gaussian_integer import
    DiscreteGaussianDistributionIntegerSampler

# Import the LWE module
from sage.crypto.lwe import LWE

# Define the Gaussian distribution over the integers with std 2.8
D = DiscreteGaussianDistributionIntegerSampler(2.8)

# Using the LWE parameters n=20, q=419, and D for the noise distribution
lwe = LWE(n=20, q=419, D=D)

# Sample 1 LWE instance c = b s + e, where s is the secret and e is the error
a, c = lwe()

# Show me a
a

# Show me c
c

# Show me the secret s by LWE
lwe._LWE__s
```

References

- [1] C. Gentry, C. Peikert, V. Vaikuntanathan. How to Use a Short Basis: Trapdoors for Hard Lattices and New Cryptographic Constructions. STOC 2008.
- [2] H. W. Lenstra, A. K. Lenstra, L. Lovász, Factoring polynomials with rational coefficients. Math. Ann. 261, 515–534, 1982.
- [3] Oded Regev. On Lattices, Learning with Errors, Random Linear Codes, and Cryptography. Journal of the ACM 56(6), 84 - 93, 2009.
- [4] C.-P. Schnorr, M. Euchner. Lattice basis reduction: improved practical algorithms and solving subset sum problems. Math. Programming 66, 181-199, 1994.
- [5] SageMath Documentation, “sage.crypto.lattice”. Retrieved from <https://doc.sagemath.org/html/en/reference/cryptography/sage/crypto/lattice.html>.
- [6] SageMath Development Team, *Discrete Gaussian Distribution on Lattices*, Available at: http://sporadic.stanford.edu/reference/stats/sage/stats/distributions/discrete_gaussian_lattice.html.
- [7] SageMath Documentation, “Free Quadratic Module (Integer Symmetric)”. Retrieved from https://doc.sagemath.org/html/en/reference/modules/sage/modules/free_quadratic_module_integer_symmetric.html.
- [8] SageMath Documentation, “Integer Dense Matrices”. Retrieved from https://doc.sagemath.org/html/en/reference/matrices/sage/matrix/matrix_integer_dense.html.
- [9] SageMath Documentation, “Integer Dense Matrices”. Retrieved from <https://doc.sagemath.org/html/en/reference/cryptography/sage/crypto/lwe.html>.