Sage for Lattice-based Cryptography

Martin R. Albrecht and Léo Ducas
Oxford Lattice School

Outline

Sage

Lattices

Rings

Sage

Blurb

Sage open-source mathematical software system

“Creating a viable free open source alternative to Magma, Maple, Mathematica and Matlab.”

Sage is a free open-source mathematics software system licensed under the GPL. It combines the power of many existing open-source packages into a common Python-based interface.

How to use it

command line run `sage`

local webapp run `sage -notebook=jupyter`

hosted webapp `https://cloud.sagemath.com` [1]

widget `http://sagecell.sagemath.org`

[1] On SMC you have the choice between “Sage Worksheet” and “Jupyter Notebook”. We recommend the latter.

Python & Cython

Sage does not come with yet-another ad-hoc mathematical programming language, it uses Python instead.

- one of the most widely used programming languages (Google, IML, NASA, Dropbox),
- easy for you to define your own data types and methods on it (bitstreams, lattices, cyclotomic rings, ...),
- very clean language that results in easy to read code,
- a huge number of libraries: statistics, networking, databases, bioinformatic, physics, video games, 3d graphics, numerical computation (SciPy), and pure mathematic
- easy to use existing C/C++ libraries from Python (via Cython)

Sage $\neq$ Python

Sage

```
1/2
```

1/2

```
2^3
```

8

```
type(2)
```

<type 'sage.rings.integer.Integer'>

Python

```
1/2
```

0

```
2^3
```

1

```
type(2)
```

<type 'int'>

Sage ≠ Python

Sage

```
type(2r)
```

```
<type 'int'>
```

```
type(range(10)[0])
```

```
<type 'int'>
```

Python

```
type(2r)
```

```
SyntaxError: invalid syntax
```

```
type(range(10)[0])
```

```
<type 'int'>
```

Files

`.sage` files are parsed as Sage code, `.py` files as Python code

```
sage: p, q = random_prime(2^512), random_prime(2^512)
sage: n = p*q
sage: ZZn = IntegerModRing(n)
```

```
sage: r = (p-1)*(q-1)
sage: ZZr = IntegerModRing(r)
```

```
sage: e = ZZ.random_element(r)
sage: while gcd(e, r) != 1:
          e = ZZ.random_element(r)
```

Naive RSA II

```
sage: type(e)
```

```
<type 'sage.rings.integer.Integer'>
```

```
sage: type(ZZr(e))
```

```
<type 'sage.rings.finite_rings.integer_mod.IntegerMod_gmp'>
```

```
sage: d = ZZr(e)^-1
sage: m = ZZn.random_element()
sage: s = m^e
sage: s^d == m
```

```
True
```

Sage has Algebraic Types I

Objects know the field, ring, group etc. where they live. We say that **elements** know their **parents**:

```
sage: parent(2)
```

```
Integer Ring
```

```
sage: K = GF(3)
sage: e = K(2)
sage: parent(e)
```

```
Finite Field of size 3
```

Sage has Algebraic Types II

Elements follow the rules of their parents:

```
sage: 2 + 3
```

```
5
```

```
sage: e, f = K(2), K(3)
sage: e + f
```

```
2
```

Sage has Algebraic Types III

If there is a canonical map between parents, it is applied implicitly

```
sage: e + 3
```

```
2
```

```
sage: v = random_vector(ZZ['x'], 2)
sage: w = random_vector(GF(7), 2)
sage: v + w
```

```
(2*x^2 + 6, 4*x + 5)
```

Sage has Algebraic Types iv

Otherwise, an error is raised:

```
sage: L = GF(5)
sage: K(2) + L(3)
```

```
TypeError: unsupported operand parent(s) for '+':
'Finite Field of size 3' and 'Finite Field of size 5'
```

See `http://doc.sagemath.org/html/en/tutorial/tour_coercion.html` for details

Sage has Algebraic Types V

Somewhat annoyingly for lattice-based cryptography, Sage likes to normalise to $[0, \ldots, q-1]$ instead of $[\lceil -q/2 \rceil, \ldots, \lfloor q/2 \rfloor]$

```
sage: K = GF(101)
sage: K(-1)
```

```
100
```

```
sage: ZZ(K(-1))
```

```
100
```

Sage has Algebraic Types VI

```
def balance(e, q=None):
    try:
        p = parent(e).change_ring(ZZ)
        return p([balance(e_) for e_ in e])
    except (TypeError, AttributeError):
        if q is None:
            try:
                q = parent(e).order()
            except AttributeError:
                q = parent(e).base_ring().order()
        return ZZ(e)-q if ZZ(e)>q/2 else ZZ(e)

balance(GF(101)(60))
balance(random_vector(GF(101), 2))
balance(PolynomialRing(GF(101), 'x').random_element(degree=3))
```

- -41
- $(-47, 31)$
- $34x^3 - 20x^2 + 11x - 48$

Sage also supports symbolic manipulation

- We define some symbols and make assumptions about them:

```
n, alpha, q, epsilon, delta_0 = var("n, alpha, q, epsilon, delta_0")
assume(alpha<1)
```

- We compute the expected norm of the shortest vector found via lattice reduction with δ_0

```
e = alpha*q/sqrt(2*pi) # stddev
m = 2*n # lattice dimension
v = e * delta_0^m * q^(n/m)  # norm of the vector
```

Symbolic Manipulation II

- Use advantage[2] $\varepsilon = \exp\left(-\pi \cdot (\|v\|/q)^2\right)$ and solve for $\log \delta_0$:

```
f = log(1/epsilon)/pi == (v/q)^2
f = f.solve(delta_0**(2*m))[0]
f = f.log().canonicalize_radical()
f = f.solve(log(delta_0))[0]
f.simplify_log()
```

$$\log\left(\delta_0\right) = \frac{\log\left(-\frac{2\,\log(\epsilon)}{\alpha^2 q}\right)}{4\,n}$$

[2] Richard Lindner and Chris Peikert. Better Key Sizes (and Attacks) for LWE-Based Encryption. In: *CT-RSA 2011*. Ed. by Aggelos Kiayias. Vol. 6558. LNCS. Springer, Heidelberg, Feb. 2011, pp. 319–339. DOI: 10.1007/978-3-642-19074-2_21.

Dense Linear Algebra

```
sage: for p in (2,3,4,7,8,9,11):
          K = GF(p, 'a')
          n = 2000 if p != 9 else 500
          A, B = (random_matrix(K, n, n) for _ in range(2))
          t = cputime()
          C = A*B
          print "%32s %10.8f"%(K,cputime(t))
```

Field	Time	Implementation
Finite Field of size 2	0.004 s	M4RI
Finite Field of size 3	0.212 s	LinBox
Finite Field in a of size 2^2	0.020 s	M4RIE
Finite Field of size 7	0.208 s	LinBox
Finite Field in a of size 2^3	0.040 s	M4RIE
Finite Field in a of size 3^2	7.28 s	generic
Finite Field of size 11	0.212 s	LinBox

Lattices

Integer Matrices

The usual operations on matrices are available:

```
sage: A = random_matrix(ZZ, 100, 100, x=-2^32, y=2^32)
sage: A*A
```

```
100 x 100 dense matrix over Integer Ring \
  (use the '.str()' method to see the entries)
```

```
sage: A = random_matrix(ZZ, 100, 100, x=-2^32, y=2^32)
sage: A.norm().log(2).n()
```

```
35.4775417878382
```

```
sage: abs(A.det()).log(2).n()
```

```
3380.14491067801
```

Bases for q-ary Lattices

We construct a basis for a q-lattice.

- We pick parameters

```
m, n, q = 5, 3, 101
```

- We compute the reduced row-echelon form of A

```
A = random_matrix(GF(q), n, m)
A.echelonize()
```

- We stack A on top of a matrix accounting for modular reductions

```
N = A.change_ring(ZZ)
S = matrix(ZZ, m-n, n).augment(q * identity_matrix(m-n))
N.stack(S, subdivide=True)
```

$$\left(\begin{array}{rrrrr} 1 & 0 & 0 & 3 & 68 \\ 0 & 1 & 0 & 4 & 96 \\ 0 & 0 & 1 & 30 & 16 \\ \hline 0 & 0 & 0 & 101 & 0 \\ 0 & 0 & 0 & 0 & 101 \end{array}\right)$$

Instance Generator

If you just want some typical lattices to play with:

```
sage: sage.crypto.gen_lattice(m=10, seed=42, type="modular")
```

```
[11  0  0  0  0  0  0  0  0  0]
[ 0 11  0  0  0  0  0  0  0  0]
[ 0  0 11  0  0  0  0  0  0  0]
[ 0  0  0 11  0  0  0  0  0  0]
[ 2  4  3  5  1  0  0  0  0  0]
[ 1 -5 -4  2  0  1  0  0  0  0]
[-4  3 -1  1  0  0  1  0  0  0]
[-2 -3 -4 -1  0  0  0  1  0  0]
[-5 -5  3  3  0  0  0  0  1  0]
[-4 -3  2 -5  0  0  0  0  0  1]
```

LLL

LLL is available. By default Sage calls `Fplll`, but you can also call `NTL`.

```
sage: A = sage.crypto.gen_lattice(m=10, seed=42, type="modular")
sage: A.LLL(delta=0.99, eta=0.51) # calls fplll
```

```
[ 0  0  1  1  0 -1 -1 -1  1  0]
[-1  1  0  1  0  1  1  0  1  1]
[-1  0  0  0 -1  1  1 -2  0  0]
[-1 -1  0  1  1  0  0  1  1 -1]
[ 1  0 -1  0  0  0 -2 -2  0  0]
[ 2 -1  0  0  1  0  1  0  0 -1]
[-1  1 -1  0  1 -1  1  0 -1 -2]
[ 0  0 -1  3  0  0  0 -1 -1 -1]
[ 0 -1  0 -1  2  0 -1  0  0  2]
[ 0  1  1  0  1  1 -2  1 -1 -2]
```

If you want LLL on Gram matrices, `Pari` is also available.

BKZ

BKZ is available. By default `Fplll` is called, but you can also call `NTL`

```
sage: A = sage.crypto.gen_lattice(m=100, seed=42, q=next_prime(2^20))
sage: B = A.BKZ(block_size=60, proof=False) # calls fplll's BKZ 2.0
sage: B[0].norm().log(2).n()
```

```
2.26178097802851
```

Note

Passing `proof=False` enables BKZ 2.0 with some decent heuristics. It will be much faster than `proof=True` which reverts back to plain BKZ without any pruning or recursive preprocessing.

Lattices

Sometimes it is more natural to work with a lattice object directly, instead of a basis matrix[3]

```
sage: from sage.modules.free_module_integer import IntegerLattice
sage: A = random_matrix(ZZ, 80, 80, x=-2000, y=2000)
sage: L = IntegerLattice(A); L
```

```
Free module of degree 80 and rank 80 over Integer Ring
User basis matrix:
80 x 80 dense matrix over Integer Ring
```

```
sage: L.shortest_vector().norm().log(2).n()
```

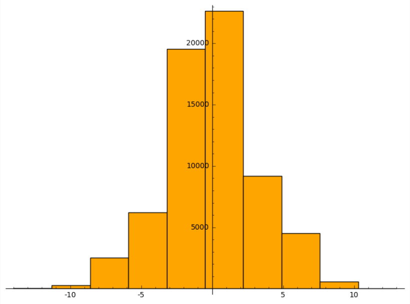
```
13.1049884393931
```

[3] Lattices are still represented by bases, though.

Discrete Gaussians: Integers

Discrete Gaussian samplers are available as:

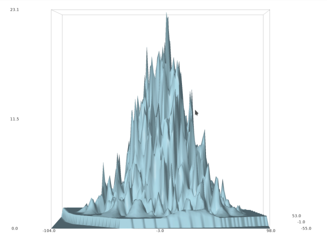
```
sage: from sage.stats.distributions.discrete_gaussian_integer import \
  DiscreteGaussianDistributionIntegerSampler
sage: D = DiscreteGaussianDistributionIntegerSampler(3.2)
sage: histogram([D() for _ in range(2^16)], color="orange")
```



Discrete Gaussians: Lattices

GPV algorithm for sampling from arbitrary lattices.[4]

```
sage: from sage.stats.distributions.discrete_gaussian_lattice import \
   DiscreteGaussianDistributionLatticeSampler
sage: A = random_matrix(ZZ, 2, 2)
sage: D = DiscreteGaussianDistributionLatticeSampler(A, 20.0)
sage: S = [D() for _ in range(2^12)]
sage: l = [vector(v.list() + [S.count(v)]) for v in set(S)]
sage: list_plot3d(l, point_list=True, interpolation='nn')
```



[4] Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In: *40th ACM STOC*. ed. by Richard E. Ladner and Cynthia Dwork. ACM Press, May 2008, pp. 197–206. DOI: 10.1145/1374376.1374407.

Learning with Errors

- Module also has `Regev` and `LindnerPeikert` samplers

```
sage: from sage.crypto.lwe import LWE
```

- We need a noise distribution sampler

```
sage: D = DiscreteGaussianDistributionIntegerSampler(3.2) # stddev
```

- We can optionally also pass in the number m of supported samples

```
sage: lwe = LWE(n=10, q=101, D=D)
```

- Get a sample and decrypt

```
sage: a,c = lwe()
sage: balance(c - a*lwe._LWE__s)
-4
```

`Fpylll` is a Python frontend for `Fplll`, giving access to its internals. It’s main aim is to facilitate experiments with lattice reduction.

```
sage: from fpylll import *
sage: A = IntegerMatrix(50, 50)
sage: A.randomize("ntrulike", bits=50, q=127)
sage: A[0].norm()
```

```
394.37418779631105
```

FPYLLL II

- We create a Gram-Schmidt object for orthogonalisation

```
sage: M = GSO.Mat(A)
sage: _ = M.update_gso()
sage: M.get_mu(1,0)
```

```
0.7982010017295588
```

- We create an LLL object that actos on `M`

```
sage: L = LLL.Reduction(M)
sage: L()
sage: M.get_mu(1,0)
```

```
0.24
```

- Operations on `M` are also applied to `A`

```
sage: A[0].norm()
```

```
5.0
```

FPYLLL: BKZ I

```
class BKZReduction:
    def __init__(self, A):
        self.A = A
        self.m = GSO.Mat(A, flags=GSO.ROW_EXPO)
        self.lll_obj = LLL.Reduction(self.m)
```

```
    def __call__(self, block_size):
        self.m.discover_all_rows()

        while True:
            clean = self.bkz_loop(block_size, 0, self.A.nrows)
            if clean:
                break
```

FPYLLL: BKZ II

```
def bkz_loop(self, block_size, min_row, max_row):
    clean = True
    for kappa in range(min_row, max_row-1):
        bs = min(block_size, max_row - kappa)
        clean &= self.svp_reduction(kappa, bs)
    return clean
```

FPYLLL: BKZ III

```
def svp_reduction(self, kappa, block_size):
    clean = True

    self.lll_obj(0, kappa, kappa + block_size)
    if self.lll_obj.nswaps > 0:
        clean = False

    max_dist, expo = self.m.get_r_exp(kappa, kappa)
    delta_max_dist = self.lll_obj.delta * max_dist

    solution, max_dist = Enumeration(self.m).enumerate(kappa, \
                             kappa + block_size, \
                             max_dist, expo, pruning=None)[0]
```

FPYLLL: BKZ IV

```
if max_dist >= delta_max_dist * (1<<expo):
    return clean

nonzero_vectors = len([x for x in solution if x])

if nonzero_vectors == 1:
    first_nonzero_vector = None
    for i in range(block_size):
        if abs(solution[i]) == 1:
            first_nonzero_vector = i
            break

    self.m.move_row(kappa + first_nonzero_vector, kappa)
    self.lll_obj.size_reduction(kappa, \
          kappa + first_nonzero_vector + 1)
```

FPYLLL: BKZ V

```
        else:
            d = self.m.d
            self.m.create_row()

            with self.m.row_ops(d, d+1):
                for i in range(block_size):
                    self.m.row_addmul(d, kappa + i, solution[i])

            self.m.move_row(d, kappa)
            self.lll_obj(kappa, kappa, kappa + block_size + 1)
            self.m.move_row(kappa + block_size, d)

            self.m.remove_last_row()

        return False
```

Rings

- Sage has polynomial rings . . .

```
sage: P = ZZ['x']; x = P.gen()
sage: P = PolynomialRing(ZZ, 'x'); x = P.gen()
sage: P, x = PolynomialRing(ZZ, 'x').objgen()
sage: P.<x> = PolynomialRing(ZZ) # not valid Python, Magma-style
```

- . . . over arbitrary rings

```
sage: R = PolynomialRing(P, 'y'); R
sage: R = PolynomialRing(IntegerModRing(100), 'y'); R
sage: R = PolynomialRing(GF(2^8,'a'), 'x'); R
```

```
Univariate Polynomial Ring in y over \
  Univariate Polynomial Ring in x over Integer Ring
Univariate Polynomial Ring in y over Ring of integers modulo 100
Univariate Polynomial Ring in x over Finite Field in a of size 2^8
```

Polynomial Rings II

- It also supports multivariate polynomial rings

```
sage: R = PolynomialRing(QQ, 'x,y'); R
sage: R.<x,y> = PolynomialRing(QQ); R
sage: R = PolynomialRing(QQ, 2, 'x'); R
sage: names = ["x%02d"%i for i in range(3)]
sage: R = PolynomialRing(IntegerModRing(100), names); R
```

```
Multivariate Polynomial Ring in x, y over Rational Field
Multivariate Polynomial Ring in x, y over Rational Field
Multivariate Polynomial Ring in x0, x1 over Rational Field
Multivariate Polynomial Ring in x00, x01, x02 \
 over Ring of integers modulo 100
```

Quotient Rings

- You can construct quotient rings:

```
sage: P.<x> = PolynomialRing(ZZ)
sage: Q = P.quotient(x^4 + 1); Q
```

```
Univariate Quotient Polynomial Ring in xbar \
  over Integer Ring with modulus x^4 + 1
```

- But I usually don't bother and do modular reductions "by hand":

```
sage: P.<x> = PolynomialRing(ZZ)
sage: f = P.random_element(degree=5); f
sage: f % (x^4 + 1)
```

```
x^5 + 9*x^4 + x^3 + x^2 + 2
x^3 + x^2 - x - 7
```

- Relative and absolute number fields are a thing:

```
sage: z = QQ['z'].0
sage: K = NumberField(z^2 - 2,'s'); K
```

```
Number Field in s with defining polynomial z^2 - 2
```

```
sage: s = K.0; s
```

```
s
```

```
sage: s^2
```

```
2
```

Cyclotomic Number Fields

Let $\mathcal{R} \simeq \mathbb{Z}[X]/(X^n + 1)$ be the ring of integers of the Cylotomic number field $\mathbb{K} = \mathbb{Q}(\zeta_m)$ for some $m = 2^k$ and $n = m/2$.

```
sage: K.<zeta> = CyclotomicField(8)
sage: OK = K.ring_of_integers()
sage: K.polynomial()
```

```
x^4 + 1
```

Cyclotomic Number Fields: Subfields

Let $\mathbb{L} = \mathbb{Q}(\zeta_{m'})$ with $m'|m$ be a subfield of $\mathbb{K}$. The ring of integers of $\mathbb{L}$ is $\mathcal{R}' \simeq \mathbb{Z}[X]/(X^{n'}+1)$ with $n' = m'/2$.

```
sage: KK, L = K.subfield(zeta^2)
sage: zeta_ = KK.gen()
sage: L(zeta_)
```

```
zeta^2
```

Cyclotomic Number Fields: Galois Group

$\mathbb{K}$ is a Galois extension of $\mathbb{Q}$, and its Galois group G is isomorphic to $\mathbb{Z}_m^*$: $i \in \mathbb{Z}_m^* \leftrightarrow (X \mapsto X^i) \in G$.

```
sage: G = K.galois_group(); G
```

```
Galois group of Cyclotomic Field of order 8 and degree 4
```

Cyclotomic Number Fields: Class Group

The first Cyclotomic field with $m = 2^k$ and a non-trivial class group is $m = 2^6$.

```
sage: K.<zeta> = CyclotomicField(2^6)
sage: K.class_number(proof=False)
```

17

Cyclotomic Number Fields: Lattices

- Converting number field elements to matrices/lattice bases:

```
sage: from sage.modules.free_module_integer import IntegerLattice
sage: f
sage: IntegerLattice(f).basis_matrix()
```

```
-10*zeta^3 + 2*zeta + 28

[ 28   2   0 -10]
[ 10  28   2   0]
[  0  10  28   2]
[ -2   0  10  28]
```

- We can use this to find small elements

```
sage: K = CyclotomicField(128)
sage: OK = K.ring_of_integers()
sage: f = OK.random_element(x=-128, y=128)
sage: L = IntegerLattice(f)
sage: _ = L.BKZ(block_size=50, proof=False)
sage: L.shortest_vector().norm().log(2).n()
```

```
9.23365749434346
```

Fin

Thank You