Lattice Reduction & Attacks

Lab

Martin R. Albrecht

27 July 2022

In this lab, we will make intensive use of FPLLL and FPyLLL.

FPLLL is a C++11 library for operating on lattices using floating point arithmetic. It implements Gram-Schmidt orthogonalisation, LLL, BKZ, BKZ 2.0[1], Slide reduction[2] and Self-Dual BKZ[3].

FPyLLL is a Python wrapper and extension of FPLLL, making its data structures and algorithms available in Python and SageMath. It also (re-)implements some algorithms in Python to make their internals easily accessible, a feature we will make use of.

G6K is C++ library & Python wrapper that implements lattice sieving. This tutorial *should* use G6K but it does not come by default with SageMath. Thus, to avoid spending all our time installing it, this lab uses only FPLLL/FPyLLL. Feel encouraged to try these exercised with G6K later, which builds on FPyLLL.

[1] Yuanmi Chen and Phong Q. Nguyen. "BKZ 2.0: Better Lattice Security Estimates". In: *ASIACRYPT 2011*. Ed. by Dong Hoon Lee and Xiaoyun Wang. Vol. 7073. LNCS. Springer, Heidelberg, Dec. 2011, pp. 1–20. DOI: 10.1007/978-3-642-25385-0_1.

[2] Nicolas Gama and Phong Q. Nguyen. "Finding short lattice vectors within Mordell's inequality". In: *40th ACM STOC*. ed. by Richard E. Ladner and Cynthia Dwork. ACM Press, May 2008, pp. 207–216. DOI: 10.1145/1374376.1374408.

[3] Daniele Micciancio and Michael Walter. *Practical, Predictable Lattice Basis Reduction*. Cryptology ePrint Archive, Report 2015/1123. https://eprint.iacr.org/2015/1123. 2015.

Introduction

In this lab, we ask you to experiment with LLL and BKZ as implemented in FPyLLL. We start with a little tutorial on how to use this library. To start, we first import the `fpylll` API into Sage's main namespace:

```
from fpylll import *
```

Integer Matrices

To experiment, we generate a q-ary lattice of dimension 100 and determinant q^{50} where q is a 30-bit prime. Before we sample our basis, we set the random seed to ensure we can reproduce our experiments later.

```
set_random_seed(1337)
A = IntegerMatrix.random(100, "qary", k=50, bits=30)
```

Remark 1. *Objects and functions in Python/Sage can be interrogated to learn more about them such as what parameters they accept (for functions) or (often) their documentation.*

Gram–Schmidt Orthogonalisation

To run LLL we have two choices. We can either run the high-level `LLL.reduction()` function or we can create the appropriate hierarchy of objects "by hand". That is, algorithms are represented by objects with which we can interact. As this exercise is about dealing with those internal objects, we are going to pursue this strategy. We, hence, first create a `MatGSO` object, which takes care of computing the Gram-Schmidt orthogonalisation. A `MatGSO` object stores the following information:

- An integral basis $\mathbf{B}$,
- the Gram–Schmidt coefficients $\mu_{i,j} = \langle \mathbf{b}_i, \mathbf{b}_j^* \rangle / \|\mathbf{b}_j^*\|^2$ for $i > j$,
- the coefficients $r_{i,j} = \langle \mathbf{b}_i, \mathbf{b}_j^* \rangle = \mu_{i,j} \cdot r_{j,j}$ for $i \geq j$

It holds that: $\mathbf{B} = \mathbf{R} \times \mathbf{Q} = (\mu \times \mathbf{D}) \times (\mathbf{D}^{-1} \times \mathbf{B}^*)$ where $\mathbf{Q}$ is orthonormal, $\mathbf{R}$ is lower triangular and $\mathbf{B}^*$ is the Gram-Schmidt orthogonalisation.

We choose the floating point type ($\approx$ bits of precision) used to represent the Gram-Schmidt coefficients as native `double`, which is fastest and fine up to dimension 170 or so. If you choose `mpfr` for arbitrary precision, you must call `FPLLL.set_precision(prec)` before constructing your object `M`, i.e. precision is global!

```
M = GSO.Mat(A, float_type="d")
```

When we said "internal", we meant it. Note that `M` is lazy, i.e. the Gram–Schmidt orthogonalisation is only computed/updated when needed. For example, as of now, none of the coefficients are meaningful:

```
M.get_r(0,0)
```

```
0.0
```

To get meaningful results, we need to trigger the appropriate computation. To compute the complete GSO, run:

```
_ = M.update_gso()
```

This is better:

```
M.get_r(0,0)/A[0].norm()^2
```

```
1.0
```

You can call `update_gso` at construction time with:

```
M = GSO.Mat(A, float_type="d", update=True)
```

Remark 2. *FP(y)LLL also supports GSO objects for Gram matrices, i.e. in lieu of a basis.*

LLL

We can now create an LLL object which operates on GSO objects. All operations performed on GSO objects, e.g. `M`, are automatically also applied to the underlying integer matrix, e.g. `A`.

```
L = LLL.Reduction(M, delta=0.99, eta=0.501, flags=LLL.VERBOSE)
```

Now that we have an LLL object, we can call it, i.e. run the algorithm. Note that you can specify a range of rows on which to perform LLL.

```
L(0, 0, 10)
```

```
Entering LLL
delta = 0.99
eta = 0.501
precision = 53
exact_dot_product = 0
row_expo = 0
early_red = 0
siegel_cond = 0
long_in_babai = 0
Discovering vector 2/10 cputime=0
Discovering vector 3/10 cputime=0
Discovering vector 4/10 cputime=0
Discovering vector 5/10 cputime=0
Discovering vector 6/10 cputime=0
Discovering vector 7/10 cputime=0
Discovering vector 8/10 cputime=0
Discovering vector 9/10 cputime=0
Discovering vector 10/10 cputime=0
End of LLL: success
```

That's maybe a bit verbose, let's continue to the end without all that feedback:

```
L = LLL.Reduction(M, delta=0.99, eta=0.501)
L()
```

If our LLL implementation is any good, then $\|\mu_{i,j}\| \leq \eta$ should hold for all $i > j$. Let's check:

```
all([abs(M.get_mu(i,j)) <= 0.501 for i in range(M.d) for j in range(i)])
```

```
True
```

We also want to check if we made progress on `A`:

```
A[0].norm()^2
```

```
57755566272.00001
```

BKZ

Calling BKZ works similarly: there is a high-level function `BKZ.reduction()` and a BKZ object `BKZ.Reduction`. However, in addition there are also several implementations of the BKZ algorithm in

```
fpylll.algorithms
```

These are re-implementations of BKZ-syle algorithms in Python which makes them rather hackable, i.e. we can modify different parts of the algorithms relatively easily. To use those, we first have to import them. We opt for BKZ 2.0:[4]

[4] See here for a simple implementation of BKZ.

```
from fpylll.algorithms.bkz2 import BKZReduction as BKZ2
```

BKZ 2.0 takes a lot of parameters, such as:

`block_size` the block size

`strategies` we explain this one below

`flags` verbosity, early abort, etc.

`max_loops` limit the number of tours

`auto_abort` heuristic, stop when the average slope of $\log(\|b_i^*\|)$ does not decrease fast enough

gh_factor heuristic, if set then the enumeration bound will be set to this factor times the Gaussian Heuristic.

It gets old fast passing these around one-by-one. Thus, FPLLL and FPyLLL introduce an object `BKZ.Param` to collect such parameters:

```
flags = BKZ.AUTO_ABORT|BKZ.MAX_LOOPS|BKZ.GH_BND
params = BKZ.Param(60, strategies=BKZ.DEFAULT_STRATEGY,
                   max_loops=4,
                   flags=flags)
```

The parameter `strategies` takes a list of "reduction strategies" or a filename for a JSON file containing such strategies. For each block size these strategies determine what pruning coefficients are used and what kind of recursive preprocessing is applied before enumeration. The strategies in `BKZ.DEFAULT_STRATEGY` were computed using fplll's `strategizer`.

```
strategies = load_strategies_json(BKZ.DEFAULT_STRATEGY)
print(strategies[60])
```

```
Strategy< 60, (40), 0.30-0.53, {}>
```

That last line means that for block size 60 we are preprocessing with block size 40 and our pruning parameters are such that enumeration succeeds with probability between 29% and 50% depending on the target enumeration radius. Still, constructing such parameter objects gets old, too, we can simply call:

```
params = BKZ.EasyParam(60, max_loops=4)
```

Finally, let's call BKZ-60 on our example lattice:

```
bkz = BKZ2(A) # or
bkz = BKZ2(GSO.Mat(A)) # or
bkz = BKZ2(LLL.Reduction(GSO.Mat(A)))

_ = bkz(params)
```

Lattice Reduction

In this exercise, we ask you to verify various predictions made about lattice reduction using the implementations available in FPyLLL.

root-Hermite factors

Recall that lattice reduction returns vectors such that

$$\|\mathbf{v}\| = \delta^{d-1} \cdot \mathrm{Vol}(\Lambda)^{1/d}$$

where δ is the root-Hermite factor which depends on the algorithm. For LLL it is $\delta_0 \approx 1.0219$ and for BKZ-k it is

$$\delta_0 \approx \left(\frac{k}{2\pi e}(\pi k)^{\frac{1}{k}}\right)^{\frac{1}{2(k-1)}}.$$

Experimentally measure root-Hermite factors for various bases and algorithms.

GS norms & Geometric series assumption

Schnorr's geometric series assumption (GSA) states that the norms of the Gram-Schmidt vectors after lattice reduction satisfy

$$\|\mathbf{b}_i^*\| = {\alpha_\beta}^{(d-1-2i)/2} \cdot \mathrm{Vol}(\Lambda)^{1/d} \text{ for some } 0 < \alpha_\beta < 1$$

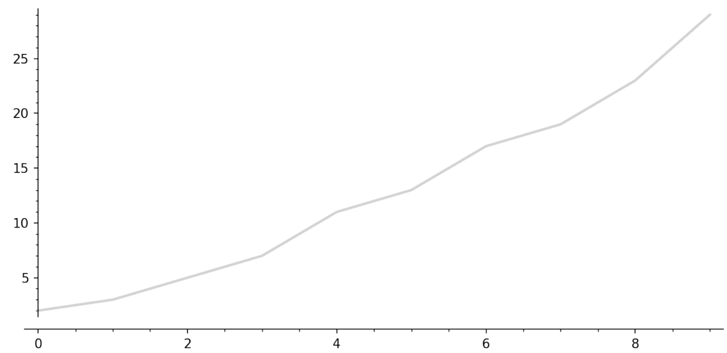
and $\alpha_\beta = \mathrm{GH}(\beta)^{1/(\beta-1)}$.

Check how well this assumption holds for various block sizes of BKZ. That is, running several tours of BKZ 2.0, plot the logs of Gram-Schmidt norms agains the GSA after each tour. You have several options to get to those norms:

- Check out the `dump_gso_filename` option for `BKZ.Param`.
- Set up BKZ parameters to run one tour only an measure between BKZ calls.
- Inherit from `fpylll.algorithms.bkz2.BKZReduction` and add the functionality to plot after each tour.

To plot you can simply call `line()` to plot, e.g.

```
kwds = {"color": "lightgrey", "dpi":150r, "thickness":2}
line(zip(range(10),prime_range(30)), **kwds)
```



Primal Attack

For varying parameters (n, q, χ_e) determine the BKZ block size required to break LWE instances corresponding to these parameters and compare your predict with experimental evidence. You may use the following lattice basis generator to run those experiments.

```
def lwe_instancef(n=20, q=7681, Xe=2, Xs=None, m=None):
    m = n if m is None else m
    Xs = Xe if Xs is None else Xs
    s = random_vector(ZZ, n, x=-Xs, y=Xs+1)
    e = random_vector(ZZ, m, x=-Xe, y=Xe+1)
    A = random_matrix(GF(q), m, n)
    b = A*s + e
    B = block_matrix(
        [
            [q*identity_matrix(ZZ, m), 0, 0],
            [A.T.lift(),identity_matrix(ZZ, n),0],
            [matrix(ZZ,1,m,b).lift(), 0, Xe],
```

```
        ])
    return B

B = lwe_instancef()
```

Example Solutions

root-Hermite factors

```
# -*- coding: utf-8 -*-
from fpylll import *

deltaf = lambda b: (b/(2*pi*e) * (pi*b)^(1/b))^(1/(2*b-1))
fmt = "n: %3d, bits: %2d, β: %2d, δ_0: %.4f, " \
    + "pred: 2^%5.2f, real: 2^%5.2f"

N = (50, 70, 90, 110, 130)
BETAS = (2, 20, 50, 60)
q = 7681

ntrials = 8
for n in N:
    for beta in BETAS:
        if beta > n:
            continue
        delta = 1.0219 if beta == 2 else deltaf(beta)
        n_pred = float(delta^(n-1) * q^(1/2))
        n_real = []
        for i in range(ntrials):
            A = IntegerMatrix.random(n, "qary", k=n/2, q=q)
            if beta == 2:
                LLL.reduction(A)
            else:
                BKZ.reduction(A, BKZ.EasyParam(block_size=beta))
            n_real.append(A[0].norm())
        n_real = sum(n_real)/ntrials
        print(fmt%(n, bits, beta, delta,
                   log(n_pred,2), log(n_real,2)))
```

```
n:  50, bits: 20, β:  2, δ_0: 1.0219, pred: 2^ 7.98, real: 2^ 7.73
n:  50, bits: 20, β: 20, δ_0: 1.0094, pred: 2^ 7.11, real: 2^ 7.40
n:  50, bits: 20, β: 50, δ_0: 1.0119, pred: 2^ 7.29, real: 2^ 7.31
n:  70, bits: 20, β:  2, δ_0: 1.0219, pred: 2^ 8.61, real: 2^ 8.53
n:  70, bits: 20, β: 20, δ_0: 1.0094, pred: 2^ 7.38, real: 2^ 7.82
n:  70, bits: 20, β: 50, δ_0: 1.0119, pred: 2^ 7.64, real: 2^ 7.56
n:  70, bits: 20, β: 60, δ_0: 1.0114, pred: 2^ 7.58, real: 2^ 7.54
n:  90, bits: 20, β:  2, δ_0: 1.0219, pred: 2^ 9.24, real: 2^ 8.96
n:  90, bits: 20, β: 20, δ_0: 1.0094, pred: 2^ 7.65, real: 2^ 8.27
n:  90, bits: 20, β: 50, δ_0: 1.0119, pred: 2^ 7.98, real: 2^ 7.93
n:  90, bits: 20, β: 60, δ_0: 1.0114, pred: 2^ 7.90, real: 2^ 7.87
n: 110, bits: 20, β:  2, δ_0: 1.0219, pred: 2^ 9.86, real: 2^ 9.62
n: 110, bits: 20, β: 20, δ_0: 1.0094, pred: 2^ 7.92, real: 2^ 8.72
n: 110, bits: 20, β: 50, δ_0: 1.0119, pred: 2^ 8.32, real: 2^ 8.26
n: 110, bits: 20, β: 60, δ_0: 1.0114, pred: 2^ 8.23, real: 2^ 8.19
n: 130, bits: 20, β:  2, δ_0: 1.0219, pred: 2^10.49, real: 2^10.41
n: 130, bits: 20, β: 20, δ_0: 1.0094, pred: 2^ 8.19, real: 2^ 9.10
n: 130, bits: 20, β: 50, δ_0: 1.0119, pred: 2^ 8.66, real: 2^ 8.64
n: 130, bits: 20, β: 60, δ_0: 1.0114, pred: 2^ 8.56, real: 2^ 8.50
```

GS norms & Geometric series assumption

`dump_gso_filename`

```
# -*- coding: utf-8 -*-
from fpylll import *

set_random_seed(1)
n, bits = 120, 40
A = IntegerMatrix.random(n, "qary", k=n/2, bits=bits)
beta = 60
tours = 8
```

```
fn = "/tmp/logs.txt"
par = BKZ.EasyParam(block_size=beta,
                dump_gso_filename=fn,
                max_loops=tours)

delta = (beta/(2*pi*e) * (pi*beta)^(1/ZZ(beta)))^(1/(2*beta-1))
alpha = delta^(-2*n/(n-1))

norms = [map(log, [(alpha^i * delta^n * 2^(bits/2))^2
                   for i in range(n)])]

BKZ.reduction(A, par)

for i, l in enumerate(open(fn).readlines()):
    if i > tours:
        break
    _norms =  l.split(":")[1] # stop off other information
    _norms = _norms.strip().split(" ") # split string
    _norms = map(float, _norms) # map to floats
    norms.append(_norms)

C = ["#4D4D4D", "#5DA5DA", "#FAA43A", "#60BD68",
           "#F17CB0", "#B2912F", "#B276B2", "#DECF3F", "#F15854"]

g  = line(zip(range(n), norms[0]), legend_label="GSA", color=C[0])
g += line(zip(range(n), norms[1]), legend_label="lll", color=C[1])

for i,_norms in enumerate(norms[2:]):
    g += line(zip(range(n), _norms),
              legend_label="tour %d"%i, color=C[i+2])
g
```

bkz.tour

```
# -*- coding: utf-8 -*-
from fpylll import *
from fpylll.algorithms.bkz2 import BKZReduction as BKZ2

set_random_seed(1)
n, bits = 120, 40
A = IntegerMatrix.random(n, "qary", k=n/2, bits=bits)
beta = 60
tours = 2
par = BKZ.EasyParam(block_size=beta)

delta = (beta/(2*pi*e) * (pi*beta)^(1/ZZ(beta)))^(1/(2*beta-1))
alpha = delta^(-2*n/(n-1))

LLL.reduction(A)

M = GSO.Mat(A)
M.update_gso()


norms  = [map(log, [(alpha^i * delta^n * 2^(bits/2))^2
                    for i in range(n)])]
norms += [[log(M.get_r(i,i)) for i in range(n)]]

bkz = BKZ2(M)

for i in range(tours):
    bkz.tour(par)
    norms += [[log(M.get_r(i,i)) for i in range(n)]]

C = ["#4D4D4D", "#5DA5DA", "#FAA43A", "#60BD68",
           "#F17CB0", "#B2912F", "#B276B2", "#DECF3F", "#F15854"]
```

```
g  = line(zip(range(n), norms[0]), legend_label="GSA", color=C[0])
g += line(zip(range(n), norms[1]), legend_label="lll", color=C[1])

for i,_norms in enumerate(norms[2:]):
    g += line(zip(range(n), _norms),
              legend_label="tour %d"%i, color=C[i+2])
g
```