Lab 2

Lattice Reduction & Attacks

Martin R. Albrecht, Guillaume Bonnoron and Léo Ducas

23 March 2017

In this lab, we will make intensive use of Fplll[1] and Fpylll[2]. Fplll is a C++11 library for operating on lattices using floating point arithmetic. It implements Gram-Schmidt orthogonalisation, LLL, BKZ, BKZ 2.0[3], Slide reduction[4] and Self-Dual BKZ[5].

Fpylll is a Python wrapper and extension of Fplll, making its data structures and algorithms available in Python and Sage (7.4 and greater). It also (re-)implements some algorithms in Python to make their internals easily accessible, a feature we will make use of.

Note: Both Fplll and Fpylll are evolving software projects. In particular, the current development versions of either library offer improvements over the latest stable released shipped with Sage. Thus, using Fpylll from within Sage and using a version compiled by yourself (outside of Sage) from GitHub will be different. We recommend, if at all possible, to use the latest development version.[6]

Both libraries being evolving software projects also means that (a) you will encounter bugs and (b) we need your help. For example, you will notice that some functions lack documentation, examples and tests. Contributions welcome![7]

Introduction

In this lab, we ask you to experiment with LLL and BKZ (2.0) 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 *
```

Note by calling `from fpylll import *` we overwite Sage's own `set_random_seed`. You can still call it as `sage.all.set_random_seed`.

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.

[1] `https://github/com/fplll/fplll`

[2] `https://github.com/fplll/fpylll`

[3] 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.

[4] 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.

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

[6] You can install the latest development versions of fplll/fpylll into Sage as well, if you run Sage locally, i.e. not on `https://sagemath.com`.

[7] `https://github.com/fplll/fplll/blob/master/CONTRIBUTING.md`

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

Reminder: 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.[8]

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 `B`,
- the Gram-Schmidt coefficients $\mu_{i,j} = \langle b_i, b_j^* \rangle / \|b_j^*\|^2$ for $i > j$,
- the coefficients $r_{i,i} = \langle b_i^*, b_i^* \rangle$ and
- the coefficients $r_{i,j} = \langle b_i, b_j^* \rangle = \mu_{i,j} \cdot r_{j,j}$ for $i > j$

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

We choose the floating point type (≈ 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 `set_precision(prec)` before constructing your object `M`, i.e. precision is global!

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

When we say "internal", we mean 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)
```

```
6.90051235708626e-310
```

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

[8] https://doc.sagemath.org/html/en/tutorial/tour_help.html

```
M.update_gso()
```

```
True
```

This is better:

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

```
1.1005727694586943e+18
1.1005727694586944e+18
```

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 in on `A`:

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

```
13340327827.0
```

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:[9]

```
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 # optionally add |BKZ.VERBOSE
par = 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

[9] Check out `https://github.com/fplll/fpylll/blob/master/src/fpylll/algorithms/simple_bkz.py` for a simple implementation of BKZ.

what kind of recursive preprocessing is applied before enumeration. The strategies in `BKZ.DEFAULT_STRATEGY` were computed using fplll's `strategizer`.[10]

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

```
'/opt/sage-devel/local/share/fplll/strategies/default.json'
Strategy< 60, (40), 0.29-0.50>
```

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.

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(par)

tours = bkz.stats.tours
map(lambda x: [x["i"], RealField(20)(x["total time"]), x["r_0"]], tours)
```

i	time	r_0
0	13.573	5729242260.0
1	25.795	3034110354.0
2	40.709	2845292328.0
3	52.384	2845292328.0

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 $\|v\| = \delta_0^d \cdot \mathrm{Vol}(L)^{1/d}$ where δ_0 is the root-Hermite factor which depends on the algorith. 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.

[10] https://github.com/fplll/strategizer

GS norms & Geometric series assumption

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

$$\|b_i^*\| = \alpha^{i-1} \cdot \|b_1\| \text{ for some } 0 < \alpha < 1.$$

Combining this with the root-Hermite factor $\|b_1\| = \delta_0^d \det(\Lambda)^{1/d}$ and $\det(\Lambda) = \prod_{i=1}^{d} \|b_i^*\|$ where d is the dimension of the lattice, we get $\alpha = \delta^{-2d/(d-1)}$.[11] Check how well this assumption holds for various block sizes of BKZ.

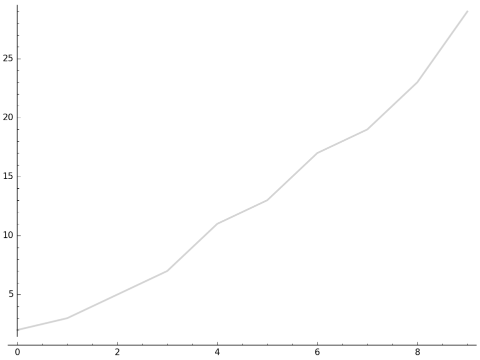
That is, running several tours of BKZ 2.0, plot the logs of Gram-Schmidt norms agains the GSA after each tour. You have several options to get to those norms:[12]

- 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 again have several options.

If you are running from within Sage, you can simply call `line()` to plot, e.g.

```
line(zip(range(10),prime_range(30)), color="lightgrey", dpi=150r, thickness=2)
```



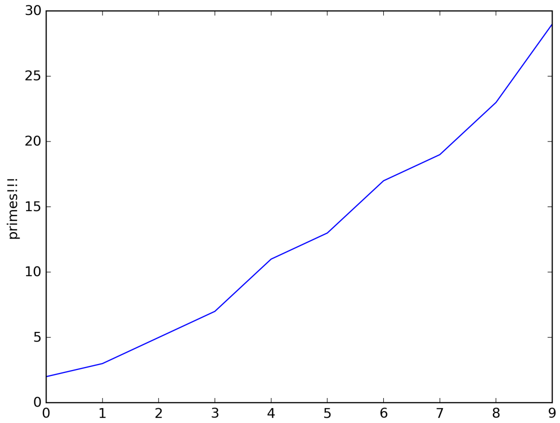
In vanilla Python, you can use matplotlib[13]

```
import matplotlib.pyplot as plt
X = range(10)
Y = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
plt.plot(X, Y)
plt.ylabel('primes!!!')
plt.savefig("lab-02-plot-line-matplotlib.png", dpi=300r, bbox_inches='tight')
plt.close()
```

[11] 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.

[12] We apologise for violating the Zen of Python so much: "There should be one — and preferably only one — obvious way to do it." `https://www.python.org/dev/peps/pep-0020/`

[13] `http://matplotlib.org`



Dual Attack

In this exercise, you are asked to break a decision-LWE instance using the dual lattice attack.

That is, assume you are given access to an lWE oracle such as

```
from sage.crypto.lwe import LindnerPeikert
lwe = LindnerPeikert(50)
```

1. Establish the BKZ block size required to distinguish with an advantage of, say, 60%.[14]

   Note that this entails finding the optimal number of samples m, i.e that m which minimises the expression $\delta_0^m \cdot q^{n/m}$.[15]

2. Construct a basis for the scaled dual lattice.[16]

3. Run BKZ on on that basis to recover a short vector v in the scaled dual lattice.

4. Use (possibly several) v to distinguish the outputs of the oracle from uniform.

5. Attempt bigger, harder instances.

6. Improve the running time by employing cleverer techniques, such as progressive BKZ[17], running in the low-advantage regime or sparse-ish re-randomisation of already reduced bases.

[14] 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.

[15] Daniele Micciancio and Oded Regev. "Lattice-based Cryptography". In: ed. by Daniel J. Bernstein, Johannes Buchmann, and Erik Dahmen. Berlin, Heidelberg, New York: Springer, Heidelberg, 2009, pp. 147–191; Martin R. Albrecht, Rachel Player, and Sam Scott. *On The Concrete Hardness Of Learning With Errors*. Cryptology ePrint Archive, Report 2015/046. http://eprint.iacr.org/2015/046. 2015.

[16] See http://doc.sagemath.org/html/en/constructions/linear_algebra.html#kernels

[17] Yoshinori Aono, Yuntao Wang, Takuya Hayashi, and Tsuyoshi Takagi. *Improved Progressive BKZ Algorithms and their Precise Cost Estimation by Sharp Simulator*. Cryptology ePrint Archive, Report 2016/146. http://eprint.iacr.org/2016/146. 2016.

Example Solutions

root-Hermite factors

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

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

ntrials = 20
for n in (50, 70, 90, 110, 130):
    for bits in (20, 40):
        for beta in (2, 20, 50, 60):
            if beta > n:
                continue
            beta = ZZ(beta)
            if beta == 2:
                delta_0 = 1.0219
            else:
                delta_0 = deltaf(beta)
            n_pred = float(delta_0^n * 2^(bits/2))
            n_real = []
            for i in range(ntrials):
                A = IntegerMatrix.random(n, "qary", k=n/2, bits=bits)
                if beta == 2:
                    LLL.reduction(A)
                else:
                    par = BKZ.Param(block_size=beta,
                                    strategies=BKZ.DEFAULT_STRATEGY,
                                    max_loops=4,
                                    flags=BKZ.MAX_LOOPS|BKZ.GH_BND)
                    BKZ.reduction(A, par)
                n_real.append(A[0].norm())
            n_real = sum(n_real)/ntrials
            print(fmt%(n, bits, beta, delta_0, log(n_pred,2), log(n_real,2)))
    print
```

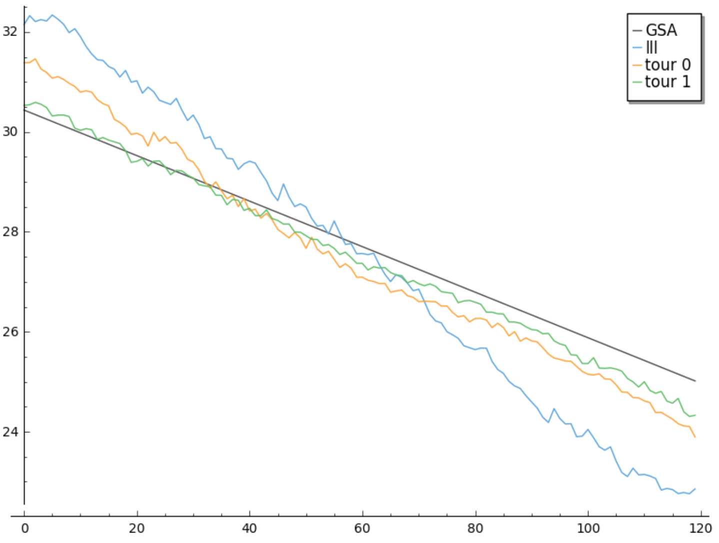
```
n:  50, bits: 20, β:  2, δ_0: 1.0219, pred: 2^11.56, real: 2^10.75
n:  50, bits: 20, β: 20, δ_0: 1.0094, pred: 2^10.67, real: 2^10.51
n:  50, bits: 20, β: 50, δ_0: 1.0119, pred: 2^10.86, real: 2^10.14
n:  50, bits: 40, β:  2, δ_0: 1.0219, pred: 2^21.56, real: 2^20.61
n:  50, bits: 40, β: 20, δ_0: 1.0094, pred: 2^20.67, real: 2^20.66
n:  50, bits: 40, β: 50, δ_0: 1.0119, pred: 2^20.86, real: 2^20.26

n:  70, bits: 20, β:  2, δ_0: 1.0219, pred: 2^12.19, real: 2^11.44
n:  70, bits: 20, β: 20, δ_0: 1.0094, pred: 2^10.94, real: 2^11.02
n:  70, bits: 20, β: 50, δ_0: 1.0119, pred: 2^11.20, real: 2^10.55
n:  70, bits: 20, β: 60, δ_0: 1.0114, pred: 2^11.14, real: 2^10.59
n:  70, bits: 40, β:  2, δ_0: 1.0219, pred: 2^22.19, real: 2^21.36
n:  70, bits: 40, β: 20, δ_0: 1.0094, pred: 2^20.94, real: 2^20.93
n:  70, bits: 40, β: 50, δ_0: 1.0119, pred: 2^21.20, real: 2^20.84
n:  70, bits: 40, β: 60, δ_0: 1.0114, pred: 2^21.14, real: 2^20.51

n:  90, bits: 20, β:  2, δ_0: 1.0219, pred: 2^12.81, real: 2^11.46
n:  90, bits: 20, β: 20, δ_0: 1.0094, pred: 2^11.21, real: 2^11.64
n:  90, bits: 20, β: 50, δ_0: 1.0119, pred: 2^11.54, real: 2^10.97
n:  90, bits: 20, β: 60, δ_0: 1.0114, pred: 2^11.47, real: 2^11.01
n:  90, bits: 40, β:  2, δ_0: 1.0219, pred: 2^22.81, real: 2^21.94
n:  90, bits: 40, β: 20, δ_0: 1.0094, pred: 2^21.21, real: 2^21.51
n:  90, bits: 40, β: 50, δ_0: 1.0119, pred: 2^21.54, real: 2^20.96
n:  90, bits: 40, β: 60, δ_0: 1.0114, pred: 2^21.47, real: 2^20.96

n: 110, bits: 20, β:  2, δ_0: 1.0219, pred: 2^13.44, real: 2^12.81
n: 110, bits: 20, β: 20, δ_0: 1.0094, pred: 2^11.48, real: 2^12.18
```

```
n: 110, bits: 20, β: 50, δ_0: 1.0119, pred: 2^11.88, real: 2^11.31
n: 110, bits: 20, β: 60, δ_0: 1.0114, pred: 2^11.79, real: 2^11.08
n: 110, bits: 40, β:  2, δ_0: 1.0219, pred: 2^23.44, real: 2^22.71
n: 110, bits: 40, β: 20, δ_0: 1.0094, pred: 2^21.48, real: 2^22.20
n: 110, bits: 40, β: 50, δ_0: 1.0119, pred: 2^21.88, real: 2^21.24
n: 110, bits: 40, β: 60, δ_0: 1.0114, pred: 2^21.79, real: 2^21.04

n: 130, bits: 20, β:  2, δ_0: 1.0219, pred: 2^14.06, real: 2^13.37
n: 130, bits: 20, β: 20, δ_0: 1.0094, pred: 2^11.75, real: 2^12.81
n: 130, bits: 20, β: 50, δ_0: 1.0119, pred: 2^12.23, real: 2^11.90
n: 130, bits: 20, β: 60, δ_0: 1.0114, pred: 2^12.12, real: 2^11.66
n: 130, bits: 40, β:  2, δ_0: 1.0219, pred: 2^24.06, real: 2^23.25
n: 130, bits: 40, β: 20, δ_0: 1.0094, pred: 2^21.75, real: 2^22.71
n: 130, bits: 40, β: 50, δ_0: 1.0119, pred: 2^22.23, real: 2^21.70
n: 130, bits: 40, β: 60, δ_0: 1.0114, pred: 2^22.12, real: 2^21.67
```

GS norms & Geometric series assumption

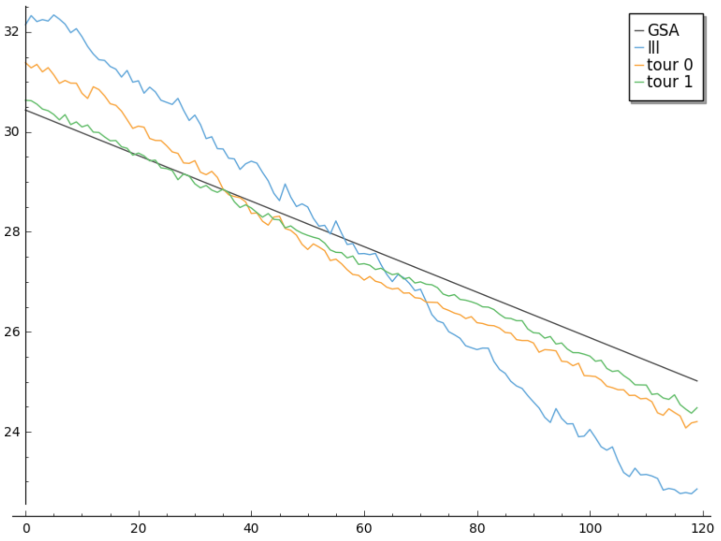
dump_gso_filename

```
# -*- 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 = 2

fn = "/tmp/logs.txt"
par = BKZ.Param(block_size=beta,
                strategies=BKZ.DEFAULT_STRATEGY,
                dump_gso_filename=fn,
                max_loops=tours)
par.flags & BKZ.MAX_LOOPS # max_loops sets flag for you

delta_0 = (beta/(2*pi*e) * (pi*beta)^(1/ZZ(beta)))^(1/(2*beta-1))
alpha = delta_0^(-2*n/(n-1))

norms = [map(log, [(alpha^i * delta_0^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)

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

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

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



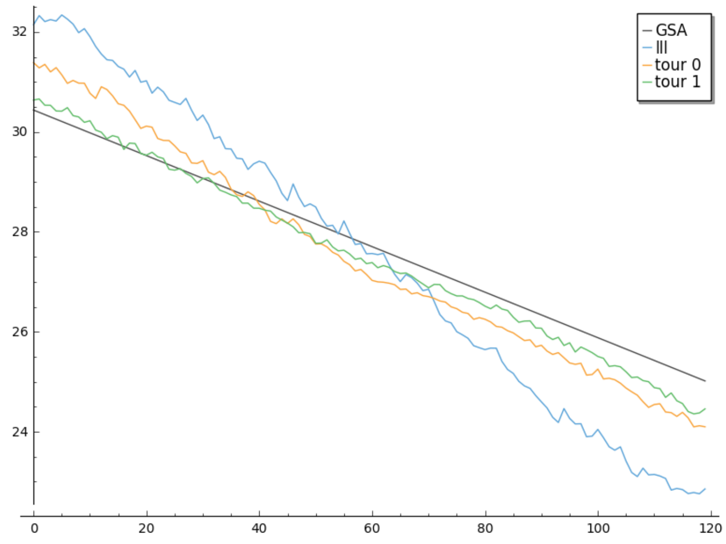
bkz.tour

```
# -*- 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.Param(block_size=beta,
                strategies=BKZ.DEFAULT_STRATEGY)

delta_0 = (beta/(2*pi*e) * (pi*beta)^(1/ZZ(beta)))^(1/(2*beta-1))
alpha = delta_0^(-2*n/(n-1))

LLL.reduction(A)

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


norms  = [map(log, [(alpha^i * delta_0^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)]]

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

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

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



MyBKZ

```
from fpylll import *
from fpylll.algorithms.bkz2 import BKZReduction as BKZ2
from fpylll.algorithms.bkz_stats import BKZStats
import time

class MyBKZ(BKZ2):
    def __call__(self, params, norms, min_row=0, max_row=-1):
        """Run the BKZ with `param`  and dump norms to ``norms``

        :param params: BKZ parameters
        :param norms: a list to append vectors of norms to
        :param min_row: start processing in this row
        :param max_row: stop processing in this row (exclusive)

        """
        # this changed in the development version of fpyll
        stats = BKZStats(self, verbose=params.flags & BKZ.VERBOSE)

        if params.flags & BKZ.AUTO_ABORT:
            auto_abort = BKZ.AutoAbort(self.M, self.A.nrows)

        cputime_start = time.clock()

        self.M.discover_all_rows()
        norms.append([self.M.get_r(j, j) for j in range(n)])

        i = 0
        while True:
            with stats.context("tour"):
                clean = self.tour(params, min_row, max_row, stats)
            norms.append([self.M.get_r(j, j) for j in range(n)])
            i += 1
            if clean or params.block_size >= self.A.nrows:
                break
            if (params.flags & BKZ.AUTO_ABORT) and auto_abort.test_abort():
                break
            if (params.flags & BKZ.MAX_LOOPS) and i >= params.max_loops:
                break
            if (params.flags & BKZ.MAX_TIME) \
               and time.clock() - cputime_start >= params.max_time:
                break

        stats.finalize()
        self.stats = stats
        return clean

set_random_seed(1)

n, bits = 120, 40
A = IntegerMatrix.random(n, "qary", k=n/2, bits=bits)
beta = 60
tours = 2
par = BKZ.Param(block_size=beta,
                strategies=BKZ.DEFAULT_STRATEGY,
                max_loops=tours)

delta_0 = (beta/(2*pi*e) * (pi*beta)^(1/ZZ(beta)))^(1/(2*beta-1))
alpha = delta_0^(-2*n/(n-1))

LLL.reduction(A)

norms  = [[(alpha^i * delta_0^n * 2^(bits/2))^2 for i in range(n)]]
bkz = MyBKZ(A)

bkz(par, norms)

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

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

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



Dual attack

We will need to invert

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

```
def block_sizef(delta_0):
    """
    Blocksize for a given delta_0
    :param delta_0: root-hermite factor

    """
    k = ZZ(40)
    RR = delta_0.parent()
    pi_r = RR(pi)
    e_r = RR(e)

    f = lambda k: (k/(2*pi_r*e_r) * (pi_r*k)**(1/k))**(1/(2*(k-1)))

    while f(2*k) > delta_0:
        k *= 2
    while f(k+10) > delta_0:
        k += 10
    while True:
        if f(k) < delta_0:
            break
        k += 1

    return k
```

We will also need to lift from mod q to the integers by mapping to the element closest to zero.

```
def balanced_lift(e):
    q = parent(e).order()
    e = ZZ(e)
    return e - q if e > q//2 else e
```

We set up our instance and recover q and α, the latter from $\sigma = \alpha \cdot q/\sqrt{2\pi}$ where σ is the **standard deviation**.

```
from sage.crypto.lwe import LindnerPeikert
adv, n = 0.6, 50

sage.all.set_random_seed(1337) # make it reproducible

lwe = LindnerPeikert(n)
q = lwe.K.order()
alpha = RR(sqrt(2*pi)*lwe.D.sigma/q)
```

We compute δ_0[18], β and m.[19]

```
log_delta_0 = log(RR(sqrt(log(1/adv)/pi))/alpha, 2)**2 / (4*n*log(q, 2))
delta_0 = RR(2**log_delta_0)

beta = block_sizef(delta_0)
m = ZZ(round(sqrt(n*log(q, 2)/log(delta_0, 2))))
beta, m
```

```
(40, 164)
```

It is time to sample from our oracle. We write the result in matrix form.

```
samples = [lwe() for _ in range(m)]

A = matrix([a for a,c in samples])
c = vector([c for a,c in samples])
```

[18] Martin R. Albrecht, Rachel Player, and Sam Scott. *On The Concrete Hardness Of Learning With Errors.* Cryptology ePrint Archive, Report 2015/046. http://eprint.iacr.org/2015/046. 2015.

[19] Daniele Micciancio and Oded Regev. "Lattice-based Cryptography". In: ed. by Daniel J. Bernstein, Johannes Buchmann, and Erik Dahmen. Berlin, Heidelberg, New York: Springer, Heidelberg, 2009, pp. 147–191.

We compute a basis for for $v \cdot A \equiv 0 \bmod q$ and extend that basis to a q-ary lattice basis B.

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

We call BKZ

```
R = B.BKZ(block_size=beta, proof=False)
```

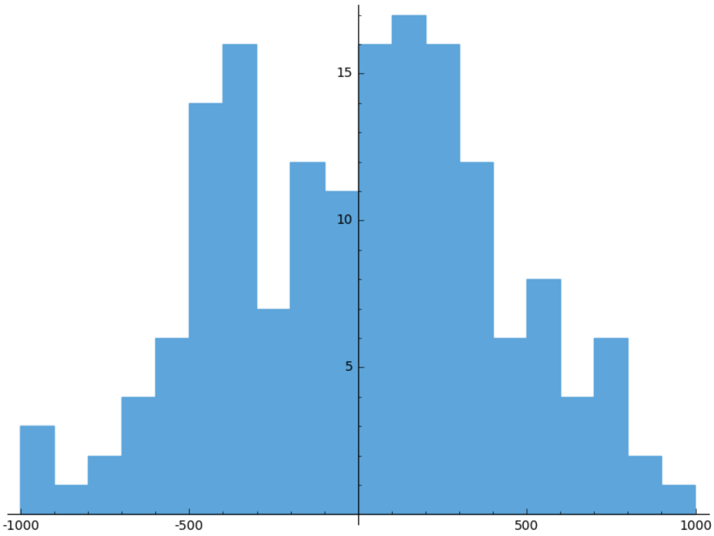
Our short vector is the first in the basis, let's look at $\langle v, c\rangle = \langle v, e\rangle$.

```
v = R[0]
balanced_lift(v*c)
```

```
-20
```

Let's #yolo and take inner products of all vectors in R.

```
l = []
for r in R.rows():
    l.append(balanced_lift(r*c))

histogram(l, color="#5DA5DA", edgecolor="#5DA5DA", bins=20)
```


15
10
5
-1000
-500
500
1000