

# FPYLLL

---

Martin R. Albrecht

20/06/2016

- `fpylll` is a Python library for performing lattice reduction on lattices over the Integers.
- It is based on the `fpLLL`
- `fpylll` also implements a few algorithms beyond `fpLLL` and provides some interface niceties

## Mission

Make implementing lattice-reduction strategies so easy, we can demand people to publish their code.

- fpylll is a Python library for performing lattice reduction on lattices over the Integers.
- It is based on the fplll
- fpylll also implements a few algorithms beyond fplll and provides some interface niceties

## Mission

Make implementing lattice-reduction strategies so easy, even I can do it.

# HIGHLEVEL INTERFACE

First of all, `fpylld` is a thin wrapper around `fpdll`. In the example below, we first generate an NTRU-like matrix and consider the norm of the first row:

```
from fpylld import IntegerMatrix, LLL
q = 1073741789
A = IntegerMatrix.random(30, "ntrulike", bits=30, q=q)
A[0].norm()
```

3294809651.09

We then call LLL reduction and observe the output:

```
LLL.reduction(A)  
A[0].norm()
```

82117.5815888

If LLL reduction isn't strong enough, we can call the BKZ algorithm for some block size  $k$ .

```
from fpylll import BKZ
BKZ.reduction(A, o=BKZ.Param(block_size=10))
A[0].norm()
```

71600.8858744

Or SVP directly.

```
from fpylll import SVP
q = 1073741789
B = IntegerMatrix.random(10, "ntrulike", bits=7, q=127)
SVP.shortest_vector(B)
```

(0, 2, -6, -6, 0, 0, 3, 2, -5, 1, 0, 5, -5, -3, 0, -1, -3, 3, -5, -7)

# LOWLEVEL INTERFACE

```
from fpylll import IntegerMatrix, GSO
q = 1073741789
A = IntegerMatrix.random(30, "ntrulike", bits=30, q=q)
M = GSO.Mat(A)
M.update_gso()
```

True

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

1.26228336805e+19



# LOWLEVEL INTERFACE

```
A[2][3] + 2*A[3][3]
```

2

```
with M.row_ops(2,4):  
    M.row_addmul(2,3,2)
```

```
A[2][3]
```

2

bkz.pyx  
enumeration.pyx  
gso.pyx  
integer\_matrix.pyx  
lll.pyx  
svpcvp.pyx  
wrapper.pyx

Of course, `fpylld` being a Python library means you can use your favourite Python libraries with it.

For example, say, we want to LLL reduce many matrices in parallel, using all our cores, and to compute the norm of the shortest vector across all matrices after LLL reduction.

We'll make use of Python's multiprocessing:

```
from multiprocessing import Pool
```

For this example, we want dimension 40, four worker processes and 32 matrices:

```
from fpylll import *  
q = 1073741789  
workers = 4  
tasks = 32  
A = []  
  
for i in range(tasks):  
    A.append(IntegerMatrix.random(20, "ntrulike", bits=30, q=q))
```

Let's get to work: we create a pool of workers and kick off the computation:

```
pool = Pool(workers)
A = pool.map(LL.reduction, A)
```

Finally, we output the minimal norm found:

```
min([A_[0].norm() for A_ in A])
```

7194.54515588

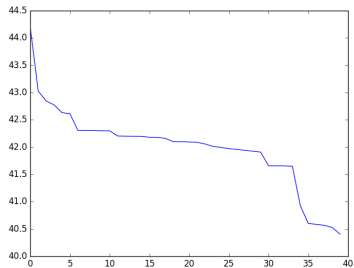
# PLOTTING

```
from fpylll import IntegerMatrix, GSO
from math import log
q = 1073741789
A = IntegerMatrix.random(40, "ntrulike", bits=30, q=q)
M = GSO.Mat(A)
M.update_gso()
gso = [log(M.get_r(i,i)) for i in range(40)]
```

True

# PLOTTING

```
import matplotlib
import matplotlib.pyplot as plt
plt.plot(gso)
plt.savefig('gso.png')
'gso.png' # return this to org-mode
```



# DEVELOPING LATTICE-REDUCTION ALGORITHMS

- The main objective of `fpylll` is to make developing and experimenting with the kind of algorithms implemented in `fplll` easier.
- For example, there are a few variants of the BKZ algorithm in the literature which essentially re-combine the same building blocks — LLL and an SVP oracle — in some way.
- These kind of algorithms should be easy to implement.

The code below is an implementation of the plain BKZ algorithm in 70 lines of Python.



```
from fpylll import IntegerMatrix, GSO, LLL, BKZ
from fpylll import Enumeration as Enum
from fpylll import gso
```

```
class BKZReduction:
    def __init__(self, A):
        wrapper = LLL.Wrapper(A)
        wrapper()

        self.A = A
        self.M = GSO.Mat(A, flags=gso.GSO.ROW_EXPO)
        self.lll_obj = LLL.Reduction(self.M)
```

```
def __call__(self, block_size):  
    self.M.discover_all_rows()  
  
    while True:  
        clean = self.tour(block_size, 0, self.A.nrows)  
        if clean:  
            break
```

```
def tour(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
```

```
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 = Enum.enumerate(self.M, max_dist, expo, \
                                         kappa, kappa + block_size, None)

    if max_dist >= delta_max_dist:
        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 + 1)
```

```
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
```

- In the meantime `fpylll` has gained a `contrib` module which implements additional algorithms.
- As of writing, it contains
  - a simple demo implementation of BKZ (see above),
  - a simple implementation of Dual BKZ and a slightly feature enhanced re-implementation of `fplll`'s BKZ which collects additional statistics compared to `fplll`'s implementation of the same algorithm.



```
from copy import copy
from fpylll import BKZ
from fpylll.contrib.bkz import BKZReduction
```

```
C = copy(A)
b = BKZReduction(C)
b(BKZ.Param(block_size=30, flags=BKZ.AUTO_ABORT|BKZ.VERBOSE))
stats = b.stats; stats
```

```
{"i": 25, "total": 12.01, "time": 0.40, "preproc": 0.10, "svp": 0.10, "r0":  
7.3483e+09, "slope": -0.0538, "enum nodes": 20.31, "max(kappa)": 6}
```

That output isn't that different from `fplll` outputs. However, in contrast to `fplll` (because I didn't bother to implement it over there, yet) we also get access to a `stats` object after the computation finished. Let's use it to inquire how many nodes were visited during enumeration

```
stats.enum_nodes
```

39977944

and how much time we spent in enumeration:

```
stats.svp_time
```

3.119223

# INTEGRATION WITH OTHER PROJECTS

`fpyl` integrates reasonably nicely with Sage: converting back and forth between data types is seamless. For example:

```
sage: A = random_matrix(ZZ, 10, 10)
sage: from fpyl import IntegerMatrix, LLL
sage: B = IntegerMatrix.from_matrix(A)
sage: LLL.reduction(B)
sage: B.to_matrix(A)[0]
```

$(-2, 1, 0, -1, 0, 0, 1, -2, 0, 0)$

## INTEGRATION WITH OTHER PROJECTS

In fact, when installed inside Sage, element access for `IntegerMatrix` accepts and returns `sage.rings.integer.Integer` directly, instead of Python integers.

```
sage: type(B[0,0])  
<type 'sage.rings.integer.Integer'>
```

# INTEGRATION WITH OTHER PROJECTS

fpylld also integrates somewhat with NumPy.

```
from fpylld import *  
A = IntegerMatrix.random(4, "ntrulike", bits=7, q=127)
```

We'd like to do some analysis on its Gram-Schmidt matrix, so let's compute it:

```
sage: M = GS0.Mat(A)  
sage: M.update_gso()
```

## INTEGRATION WITH OTHER PROJECTS

Let's dump it into a NumPy array and spot check that the result is reasonably close:

```
import numpy
from fpylll.numpy import dump_mu
N = numpy.ndarray(dtype="double", shape=(8,8))
dump_mu(N, M, 0, 8)
N[1,0] - M.get_mu(1,0)
```

0.0

Let's do something more or less useful with our output:

```
numpy.linalg.eigvals(N)
```

```
[ -7.99122854e-40 +0.00000000e+00j -5.65065189e-40
+5.65065189e-40j -5.65065189e-40 -5.65065189e-40j -8.15663058e-56
+7.99122854e-40j -8.15663058e-56 -7.99122854e-40j 7.99122854e-40
+0.00000000e+00j 5.65065189e-40 +5.65065189e-40j 5.65065189e-40
-5.65065189e-40j]
```

# TESTS

`fpylll` runs tests on every check-in for Python 2 and 3. As an added benefit, this extends test coverage for `fpdll` as well, which only has a few highlevel tests.

```
def test_lll_lll():
    for m, n in dimensions:
        A = make_integer_matrix(m, n)
        b00 = []
        for float_type in float_types:
            B = copy(A)
            M = GSO.Mat(B, float_type=float_type)
            lll = LLL.Reduction(M)
            lll()
            if (m, n) == (0, 0):
                continue
            b00.append(B[0, 0])
        for i in range(1, len(b00)):
            assert b00[0] == b00[i]
```

“This is all nice and well”, I hear you say, “but I prefer to do my computations in Lisp, so thanks, but not thanks”.

[https://imgs.xkcd.com/comics/lisp\\_cycles.png](https://imgs.xkcd.com/comics/lisp_cycles.png)

No worries, Hy has you covered:

```
=> (import [fpylll [*]])
=> (setv q 1073741789)
=> (setv A (.random IntegerMatrix 30 "ntrulike" :bits 30 :q q))
=> (car A)
row 0 of <IntegerMatrix(60, 60) at 0x7f1cbbfbf888>
=> (get A 1)
row 1 of <IntegerMatrix(60, 60) at 0x7f1cbbfbf888>
=> (-> (car A) (.norm))
4019682565.5285482

=> (.reduction LLL A)
=> (.norm (car A))
6937.9845776709535
```



# HELP WANTED

- `fpylll` isn't quite done yet.
- Besides testing and documentation, it would be nice if someone would attempt to re-implement `fpLLL`'s LLL wrapper in pure Python.
- This would serve as a test case to see if everything that's needed really is exposed and as a starting point for others who like to tweak the strategy.
- Speaking of LLL, `fpylll` is currently somewhat biased towards playing with BKZ, i.e. it would be nice to see how useful it is for trying out tweaks to the LLL algorithm.

Thank You