FPLLL

FPyLLL

Martin R. Albrecht

2017/07/06

Outline

FPYLLL

- `fpylll` is a Python (2 and 3) 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

https://github.com/fplll/fpylll

A Mission Statement

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

A Mission Statement

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

… and make it easy for everyone else in the process, too.

Why

"Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: ***premature optimization is the root of all evil.*** *Yet we should not pass up our opportunities in that critical 3%." — Donald Knuth*

Why

"Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: ***premature optimization is the root of all evil.*** *Yet we should not pass up our opportunities in that critical 3%." — Donald Knuth*

2015, a mild autumn afternoon in Bochum

Léo Wouldn't it be great if we could play with lattice reduction in Python

Martin Hold my beer . . .

How

Python

- Python is a nice, high-level language commonly used for computational mathematics (NumPy, SageMath, . . .)
- It is, however, not very fast.
- Yet, many lattice-reduction algorithms or algorithms calling lattice-reduction string together lower-level but long-ish running algorithms (LLL, enumeration, Gram-Schmidt orthogonalisation)

We don't need the performance of C++ everywhere. At higher levels, expressiveness and ease-of-use beat raw performance.[1]

[1] Okay, to be fair modern C++11 looks kinda like Python, but there's still the compile-and-run cycle.

Cython

Cython[2] is an optimising static compiler for both the Python programming language and the extended Cython programming language.

- Write Python code that calls back and forth from and to C or C++ code natively at any point.
- Easily tune readable Python code into plain C performance by adding static type declarations.
- Integrate natively with existing code and data from legacy, low-level or high-performance libraries and applications.

[2] http://cython.org

Dependencies

fpylll relies on the following C/C++ libraries:

- GMP or MPIR for arbitrary precision integer arithmetic.
- MPFR for arbitrary precision floating point arithmetic.
- QD for double double and quad double arithmetic (optional).
- fplll for pretty much everything.

fpylll also relies on

- Cython for linking Python and C/C++.
- cysignals for signal handling such as interrupting C++ code.
- py.test for testing Python.
- flake8 for linting.

We also suggest

- IPython for interacting with Python
- Numpy for numerical computations

Getting it I

1. Create a new virtualenv and activate it:

```
$ virtualenv env
$ source ./env/bin/activate
```

2. Install the required libraries - GMP or MPIR and MPFR - if not available already. You may also want to install QD.
3. Install fplll:

```
$ (fpylll) ./install-dependencies.sh $VIRTUAL_ENV
```

4. Install Cython and Python requirements:

```
$ (fpylll) pip install Cython
$ (fpylll) pip install -r requirements.txt
```

Getting it II

5. If you are so inclined, run:

```
$ (fpylll) pip install -r suggestions.txt
```

   to install suggested Python packages as well.

6. Build fpylll:

```
$ (fpylll) export PKG_CONFIG_PATH="$VIRTUAL_ENV/lib/pkgconfig"
$ (fpylll) python setup.py build
$ (fpylll) python setup.py install
```

7. To run fpylll, you will need to:

```
$ (fpylll) export LD_LIBRARY_PATH="$VIRTUAL_ENV/lib"
```

   so that Python can find fplll and friends.

8. Start Python:

```
$ (fpylll) ipython
```

To reactivate the virtual environment later:[3]

```
$ source ./env/bin/activate
export LD_LIBRARY_PATH="$VIRTUAL_ENV/lib"
```

Alternatives

fpylll is also available via PyPI, Conda-Forge for Conda and in SageMath.

[3] See `https://github.com/fplll/fpylll` for how to automate the `export` step.

Implementation

Declaration

Declaring C++ classes

```
# fpylll/fplll/fplll.pxd

cdef extern from "fplll/nr/matrix.h" namespace "fplll":
    cdef cppclass ZZ_mat[T]:

        ZZ_mat()
        ZZ_mat(int r, int c)

        …

        int get_cols() nogil
```

Declaring Cython classes

```
# fpylll/fplll/integer_matrix.pxd

from fpylll.gmp.types cimport mpz_t
from fplll cimport ZZ_mat

cdef class IntegerMatrix:
    cdef ZZ_mat[mpz_t]  *_core
```

Implementation (Constructor)

```
# fpylll/fplll/integer_matrix.pyx

from fpylll.gmp.types cimport mpz_t
from fplll cimport ZZ_mat

cdef class IntegerMatrix:
    def __init__(self, arg0, arg1=None):
        cdef int i, j

        if PyIndex_Check(arg0) and PyIndex_Check(arg1):
            if arg0 < 0:
                raise ValueError("Number of rows must be >0")

            if arg1 < 0:
                raise ValueError("Number of columns must be >0")

            self._core = new ZZ_mat[mpz_t](arg0, arg1)
            return
        …

        else:
            raise TypeError("Parameters arg0 and arg1 not understood")
```

Implementation (Method)

```
# fpylll/fplll/integer_matrix.pyx

@property
def ncols(self):
    """Number of Columns

    :returns: number of columns

    >>> from fpylll import IntegerMatrix
    >>> IntegerMatrix(10, 10).ncols
    10

    """
    return self._core.get_cols()
```

Catching Errors and Interrupts

Errors and `abort()` calls do not have to crash your Python shell. You can also interrupt long running computations.

```
# fpylll/fplll/lll.pyx

from cysignals.signals cimport sig_on, sig_off

sig_on()
self._core.mpz_double.lll(kappa_min, kappa_start, kappa_end, \
                          size_reduction_start)
r = self._core.mpz_double.status
sig_off()
```

Dark Side: Declaration

```
# fpylll/fplll/decl.pxd

IF HAVE_QD:
    ctypedef union mat_gso_core_t:
        MatGSO[Z_NR[mpz_t], FP_NR[double]] *mpz_double
        MatGSO[Z_NR[mpz_t], FP_NR[longdouble]] *mpz_ld
        MatGSO[Z_NR[mpz_t], FP_NR[dpe_t]] *mpz_dpe
        MatGSO[Z_NR[mpz_t], FP_NR[dd_real]] *mpz_dd
        MatGSO[Z_NR[mpz_t], FP_NR[qd_real]] *mpz_qd
        MatGSO[Z_NR[mpz_t], FP_NR[mpfr_t]] *mpz_mpfr
ELSE:
    ctypedef union mat_gso_core_t:
        MatGSO[Z_NR[mpz_t], FP_NR[double]] *mpz_double
        MatGSO[Z_NR[mpz_t], FP_NR[longdouble]] *mpz_ld
        MatGSO[Z_NR[mpz_t], FP_NR[dpe_t]] *mpz_dpe
        MatGSO[Z_NR[mpz_t], FP_NR[mpfr_t]] *mpz_mpfr
```

Dark Side: Implementation

```
# fpylll/fplll/gso.pyx

@property
def d(self):
    if self._type == mpz_double:
        return self._core.mpz_double.d
    IF HAVE_LONG_DOUBLE:
        if self._type == mpz_ld:
            return self._core.mpz_ld.d
    if self._type == mpz_dpe:
        return self._core.mpz_dpe.d
    IF HAVE_QD:
        if self._type == mpz_dd:
            return self._core.mpz_dd.d
        if self._type == mpz_qd:
            return self._core.mpz_qd.d
    if self._type == mpz_mpfr:
        return self._core.mpz_mpfr.d

    raise RuntimeError("MatGSO object '%s' has no core."%self)
```

What

FPLLL Modules

IntegerMatrix matrices over `mpz_t` but not over `long`

GSO complete API for plain Gram-Schmidt objects, all floating point types, not Gram variant

LLL complete API (?)

BKZParam complete API

BKZ only high-level `reduction` routine

Wrapper high-level `reduction` routine

Enumeration complete API (?)

Pruner complete API (?)

GaussSieve complete API (?)

SVP complete API (?)

CVP complete API (?)

Extended API for Integer Matrices

`mul` naive matrix $\times$ matrix products

`mod` apply modular reduction modulo q to a matrix

`apply_transform` apply transformation matrix U to a matrix.

`submatrix` construct a new submatrix

`multiply_left` $v \cdot A$

Extended API for GSO

`from_canonical` Given a vector $\mathbf{v}$ wrt the canonical basis $\mathbb{Z}^n$ return a vector wrt the Gram-Schmidt basis $\mathbf{B}^*$

`to_canonical` Given a vector $\mathbf{v}$ wrt the Gram-Schmidt basis $\mathbf{B}^*$ return a vector wrt the canonical basis $\mathbb{Z}^n$

`babai` Return lattice vector close to $\mathbf{v}$ using Babai's nearest plane algorithm

New Modules

Have:

BKZStats	collecting trees of statistics for BKZ-like algorithms
SimpleBKZ	simple, proof-of-concept implementation of BKZ2
SimpleDBKZ	simple, proof-of-concept implementation of Self-Dual BKZ
BKZ2	feature-complete re-implementation of BKZ as implemented in fplll

Want:

DBKZ	a re-implementation of the full Self-Dual BKZ in Python
Wrapper	a re-implementation of the fplll LLL wrapper in Python
	???

Simple BKZ I

We need to import some modules

```
from __future__ import absolute_import  # Python 3
from fpylll import IntegerMatrix, GSO, LLL, BKZ
from fpylll import Enumeration
```

We need a `GSO` object and an `LLL` object

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

Simple BKZ II

BKZ simply runs tours aka looks until nothing changes or the abort condition is met.

```
    def __call__(self, block_size):
        auto_abort = BKZ.AutoAbort(self.m, self.A.nrows)

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

Simple BKZ III

A tour simply proceeds index by index and records if something changed

```
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
```

Simple BKZ IV

Preprocessing

```
    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
```

Enumeration

```
        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]

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

Insert found vector into basis

```
        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
```

Tests

fpylll runs tests on every check-in for Python 2 and 3. As an added benefit, this extends test coverage for fplll as well.

```
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]
```

Multicore

Of course, fpylll 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.

Multicore

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(40, "qary", q=q, k=20))
```

Multicore

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

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

Finally, we output the minimal norm found:

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

7194.54515588

Sage Integration I

fpylll 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 fpylll 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)

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'>
```

Contributing

Contributing

Yes, please!

All contributions to `fpylll`

- are automatically tested using `py.test`
- must follow the coding style

Project ideas

- extend interface to cover LLL on Gram Matrices
- check API coverage of `fplll`
- function-level and high-level documentation
- automated attacks/scripts for challenges (SVP, LWE, NTRU)
- port API extensions down to `fpylll`

Fin

Thank You