Documentation for `matrices test.py`

Francesca Bianchi

September 10, 2015

Abstract

The following documentation aims to give an overview of what different operations can be done with the file `matrices test.py`. These include operations among some special infinite upper triangular matrices, also in the case in which there are some unknown upper diagonals. In particular, the file introduces two different classes, `M` and `MX`, and defines functions which take arguments from both or either of the classes.

A quick reference

This part provides a quick guide on how to use the file. For a more detailed explanation refer to the later sections.
The file allows to do operations with elements of a particular group. Consider

$$\Gamma =< x_0, ..., x_6 | x_i x_{i+1} x_{i+3} = \mathrm{Id} >,$$

where Id denotes the identity element.
There is a faithful representation of Γ in the group of finite band upper triangular infinite matrices with entries in $M(3, \mathbb{F}_2)$, identities on the diagonal and entries on the upper diagonals with periodicity 3 ($g_{ij} = g_{i+3,j+3}$ for all $i, j \geq 1$ for any g in this group) (see [1]).
Each element in Γ may thus be identified with an infinite matrix of this type.
The generators are all built-in and can be called by `x0`,..., `x6`.
An upper diagonal can be described equivalently by a 3×9 matrix with entries in $\mathbb{F}_2$ or by a 3-tuple of non-negative numbers, each less than or equal to 511. Indeed, if the first 3 entries on an upper diagonal are $a_1, a_2, a_3 \in M(3, \mathbb{F}_2)$, the 3×9 matrix $[a_1, a_2, a_3]$ will describe the upper diagonal entirely, because of the periodicity 3. Moreover, for $k = 1, 2, 3$, the matrix $((a_k)_{ij})_{1 \leq i,j \leq 3}$ can be represented by the number $A_k = 256(a_k)_{11} + 128(a_k)_{12} + 64(a_k)_{13} + 32(a_k)_{21} + 16(a_k)_{22} + 8(a_k)_{23} + 4(a_k)_{31} + 2(a_k)_{32} + (a_k)_{33}$. Therefore, $[A_1, A_2, A_3]$ describes the same upper diagonal as $[a_1, a_2, a_3]$.
For instance,

```
>>> x0
M_0 ([[11, 11, 11], [17, 17, 17], [26, 26, 26], [11, 11, 0], [17, 0, 0]])
>>> print x0
M_0 ([matrix([[0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 1, 0, 0, 1, 0, 0, 1],
        [0, 1, 1, 0, 1, 1, 0, 1, 1]]), matrix([[0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 1, 0, 0, 1, 0, 0, 1, 0],
        [0, 0, 1, 0, 0, 1, 0, 0, 1]]), matrix([[0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 1, 1, 0, 1, 1, 0, 1, 1],
        [0, 1, 0, 0, 1, 0, 0, 1, 0]]), matrix([[0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 1, 0, 0, 1, 0, 0, 0],
        [0, 1, 1, 0, 1, 1, 0, 0, 0]]), matrix([[0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 1, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 1, 0, 0, 0, 0, 0, 0]])])
```

Here the subscript 0 indicates that there are no zero upper diagonals before the one described by $[11, 11, 11]$. Inside the brackets we find the description of the subsequent non-zero upper diagonals. All the diagonals after $[17, 0, 0]$ are zero.
We can multiply and take powers. For example, $x_3 x_6^{-1} x_5^2$ would be

```
>>> x3*(x6**(-1))*(x5**2)
M_0 ([[57, 164, 83], [40, 200, 491], [1, 220, 460], [56, 465, 24], [20, 146, 369],
[3, 398, 430], [35, 162, 131], [3, 73, 256], [10, 276, 133], [36, 511, 195],
[58, 24, 390], [3, 48, 325], [6, 40, 0], [5, 0, 0]])
```

We can also work with elements of which only some upper diagonals are known. For instance,

```
>>> a=M(12,[26,26,26])
>>> b=MX(2,matrix([[0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 1, 0, 0, 1, 0, 0, 1],
[0, 1, 1, 0, 1, 1, 0, 1, 1]]))
>>> a
M_12 ([[26, 26, 26]])
>>> b
M_2 ([[11, 11, 11]],?)
>>> a.comm(b)
M_16 ([],?)
>>> a.conj(b)
M_2 ([[11, 11, 11]],?)
```

Here `a.comm(b)` and `a.conj(b)` give $a^{-1}b^{-1}ab$ and $a^{-1}ba$ respectively.
We can also produce an element from other two by adding their upper diagonals.

```
>>> a+b
M_2 ([[11, 11, 11]],?)
```

Finally, we can truncate elements in the following way:

```
>>> c=x0**2
>>> c
M_1 ([[26, 26, 26], [0, 0, 0], [17, 17, 17], [0, 26, 26], [11, 11, 11],
[17, 17, 0], [26, 0, 0]])
>>> c.trunc(5)
M_1 ([[26, 26, 26], [0, 0, 0], [17, 17, 17], [0, 26, 26]],?)
```

A more detailed guide

1 Useful procedures

The first part of the code defines some useful operations.

`inversebigmatrix(B,q)`:
Let B be an upper triangular square matrix of arbitrary dimension, whose entries are 3×3 matrices and whose diagonal entries are all equal to the identity. Let q be a positive integer. Then `inversebigmatrix(B,q)` returns the inverse of B modulo q.

`transfmn(M)`:
Let $M = (M_{ij})_{1 \leq i,j \leq 3}$ be a 3×3 matrix with entries in $\mathbb{F}_2$. Then `transfmn(M)` returns the integer $256M_{11} + 128M_{12} + 64M_{13} + 32M_{21} + 16M_{22} + 8M_{23} + 4M_{31} + 2M_{32} + M_{33}$.

`transfnm(n)`:
Given an integer $1 \leq n \leq 511$, `transfnm(n)` returns the unique 3×3 matrix M with entries in $\mathbb{F}_2$ such

that `transfmn(M) == n`.

`extract(k,M)`:
Let $M = (M_{ij})_{1\leq i\leq 3,\ 1\leq j\leq 9}$ be a 3×9 matrix.
If k is an integer such that $1 \leq k \leq 3$, `extract(k,M)` returns the 3×3 matrix $(M_{ij})_{1\leq i\leq 3,\ 3k-2\leq j\leq 3k}$.
For all other choices of k, the function returns:
`'You entered a value of k out of range:  k must be an integer between 1 and 3'`.

`comb(U1,U2,U3)`:
Given the 3×3 matrices U_1, U_2, U_3, `comb(U1,U2,U3)` returns the unique 3×9 matrix U such that `extract(i,M) == Ui`, for $1 \leq i \leq 3$.

`numm(ss)`:
The function `numm` takes a list ss of three non-negative integers less than or equal to 511 and returns the 3×9 matrix `comb(transfnm(ss[0]),transfnm(ss[1]),transfnm(ss[2]))`.

`matt(M)`:
The function `matt` is the inverse of `numm`: it takes a 3×9 matrix, reduces it modulo 2, and returns the corresponding 3-tuple of integers.

`p2(n)`:
Given a positive integer n, `p2(n)` returns $\max\{m : 2^m \leq n\}$.

`impp2(n)`:
Given a positive integer n, `impp2(n)` returns the unique list $[i_i, ..., i_k]$, with $i_1 > i_2 > ... > i_{k-1} > i_k$ and $n = 2^{i_1} + ... + 2^{i_k}$.

The matrices $\alpha_1, \beta_1, \gamma_1, \alpha_2, \beta_2, \gamma_2, \alpha_3, \beta_3$ defined in the Proof of Proposition 2.5 in [1] are all built-in and in the final part of the code the following function is defined.

`lincomb(M)`:
Given a 3×9 matrix M over $\mathbb{F}_2$, `lincomb(M)` returns M as a linear combination of $\alpha_1, \beta_1, \gamma_1$ or $\alpha_2, \beta_2, \gamma_2$ or α_3, β_3 if M belongs to the $\mathbb{F}_2$-span of one of these 3 set of matrices and returns no output otherwise. The argument M may also be replaced by the corresponding 3-tuple of numbers.

Example 1.

```
>>> A=matrix([[0,1,1],[1,0,0],[1,1,1]])
>>> transfmn(A)
231
>>> transfnm(231)
matrix([[0, 1, 1],
        [1, 0, 0],
        [1, 1, 1]])
>>> print alpha1
[[0 0 0 0 1 1 0 1 0]
 [0 1 0 1 0 0 0 0 1]
 [1 1 1 0 0 0 0 1 0]]
>>> B=extract(2,alpha1)
>>> B
matrix([[0, 1, 1],
        [1, 0, 0],
        [0, 0, 0]])
>>> comb(A,B,transfnm(0))
matrix([[0, 1, 1, 0, 1, 1, 0, 0, 0],
        [1, 0, 0, 1, 0, 0, 0, 0, 0],
        [1, 1, 1, 0, 0, 0, 0, 0, 0]])
```

```
>>> numm([1,0,231])
matrix([[0, 0, 0, 0, 0, 0, 0, 1, 1],
        [0, 0, 0, 0, 0, 0, 1, 0, 0],
        [0, 0, 1, 0, 0, 0, 1, 1, 1]])
>>> matt(alpha1)
[23, 224, 138]
>>> p2(10)
3
>>> impp2(10)
[3, 1]
```

Example 2.

```
>>> A=(alpha1+beta1) %2
>>> lincomb(A)
'alpha1+beta1'
>>> >>> lincomb([11,11,11])
'alpha1+gamma1'
>>> lincomb([0,0,0])
'0'
```

2 The class `M`

2.1 Instances of `M`

An instance g in this class represents an infinite upper triangular matrix with the following properties:

1. Each entry is a 3×3 matrix over $\mathbb{F}_2$;
2. Each diagonal entry is the identity;
3. $g_{ij} = g_{i+3,j+3}$ for all $i, j \geq 1$;
4. There exists $n \geq 1$ such that $g_{ij} = \mathbf{0}$ for all i, j with $j - i \geq n$, where $\mathbf{0}$ denotes the 3×3 zero matrix.

Because of Property 3, we may define an upper diagonal by a 3×9 matrix (see [1]).
We define an element g in `M` in the following way. Let $U_1, ..., U_m$ be 3×9 matrices and k an integer. Then `g=M(k,U1,...,Um)` defines the element of `M` with k zero upper diagonals followed by m diagonals described by $U_1, ..., U_m$. The matrices $U_1, ..., U_m$ may be replaced by the corresponding 3-tuples of numbers.

`print g`:
Assume neither U_1 nor U_m is the zero matrix. Then the command `print g` gives `M_k([U1,...,Um])` and we can call k by `g.k` and $[U_1, ..., U_m]$ by `g.m`.
Assume now that $U_i = 0$ for all $i \leq j$, for some $1 \leq j < m$ and that both U_{j+1} and U_m are non zero. Then `print g` gives `M_k+j([Uj+1,...,Um])`.
Similarly, if $U_i = 0$ for all $i \geq j$ for some $1 < j \leq m$ and neither U_0 nor U_{j-1} is zero, then `print g` gives `M_k([U1,...,Uj-1])`.
Finally, if we enter `g=M(k,)`, for some $k \geq 0$, or if U_i is the zero 3×9 matrix for all i, then `print g` gives `Id`.

`g`:
Just typing `g` in the shell gives the same output as `print g`, but with the 3×9 matrices replaced by 3-tuples of numbers. It is the same as `print g.transfmn()` (see 2.2). Note, however, that while the latter may be used also in the file, the command `g` gives an output only if typed in the shell.

The generators $x_0, x_1, x_2, x_3, x_4, x_5, x_6$ are all built-in.

Example 3.

```
>>> x0
M_0 ([[11, 11, 11], [17, 17, 17], [26, 26, 26], [11, 11, 0], [17, 0, 0]])
>>> len(x0.m)
5
>>> g=M(3,[0,0,0],[1,2,3],[0,0,0])
>>> print g
M_4 ([matrix([[0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 1, 0, 1, 0, 0, 1, 1]])])
>>> g.k
4
>>> g.m
[matrix([[0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 1, 0, 1, 0, 0, 1, 1]])]
>>> h=M(3,[0,0,0],[0,0,0])
>>> print h
Id
```

2.2 Operations within the class `M`

`__eq__()` and `__ne__()`:
We can compare instances in `M` in the obvious way.
For g, h in `M`, `g==h` (resp. `g!=h`) returns `True` (resp. `False`) if g and h represent the same matrix and `False` (resp. `True`) otherwise.

`transfmn()`:
Given an instance `g=M(k,U1,...,Um)` of `M`, the command `g.transfmn()` returns
`M_k([[n11,n12,n13],...,[nm1,nm2,nm3]])`, where $0 \leq n_{ij} \leq 511$ represents the matrix j of U_i (see [2]).

`ext(n,m)`:
Given g in `M` and n, m positive integers, `g.ext(n,m)` is the $n \times m$ matrix obtained from the first n rows and first m columns of the infinite matrix represented by g.

`__mul__()`:
Given g, h in `M`, `g*h` returns $g \cdot h$.

`inv()`:
`g.inv()` returns the inverse of g. If the precision parameter is large enough, the output of `print g.inv()` is an element in `M`; otherwise, it is an element in `MX` (see Section 3).
The precision parameter is set by default to be equal to 1 and can be modified by overwriting the global variable `precinv` (see below). Note that the value of `precinv` is changed locally inside some functions in order not to lose information (for instance in `conj` and `comm` if at least one of the two arguments is in `MX`). However its value is then reset to the original value.

`precinv`:
As mentioned above, the global variable `precinv` controls the maximum number of upper diagonals we allow to be computed in the inverse. Suppose g has $k(g)$ zero upper diagonals and $m(g)$ non-zero upper diagonals. Then it can be shown that g^{-1} has $k(g)$ zero upper diagonals. Let $m(g^{-1})$ be the number of non-zero upper diagonals of g^{-1}. When we type `g.inv()`, the function `inv()` will find the exact inverse of g if $k(g) + m(g^{-1}) \leq \text{precinv} \cdot (k(g) + m(g))$ and it will return the first $\text{precinv} \cdot (k(g) + m(g))$ upper diagonals of g^{-1} otherwise.
If the program is used to do operations only involving the generators, it is recommendable to set `precinv`

to 1.

`__pow__()`:
Let n be an integer (possibly zero or negative). Then `g**n` returns g^n. In particular, note that the inverse of g is returned both if we type `g.inv` or `g**(-1)`.

`conj()`:
`g.conj(h)` returns $g^{-1}hg$ (it may be in `MX` if `g**(-1)` is in `MX`).

`comm()`:
`g.comm(h)` returns $g^{-1}h^{-1}gh$ (it may be in `MX` if `g**(-1)` or `h**(-1)` is in `MX`).
`comm()` also computes higher commutators. That is: `g.comm(y0,...,yj)` returns the higher commutator $[g, y_0, ..., yj] = [[..[g, y_0], ..., y_{j-1}], y_j]$.

`commr()`:
`g.comm(y0,...,yj)` returns the higher commutator $[g, y_0, ..., yj] = [g, [y_0, ..., [y_{j-1}, y_j]..]$. Note that `g.commr(h)` is the same as `g.comm(h)`.

`__add__()`:
`g+h` returns the instance of `M` representing the matrix whose upper diagonals are the sum of the upper diagonals of g and h. Note that this is not a proper sum, in that `g+h` has identities on the diagonal instead of zero matrices.

`trunc(n)`:
`g.trunc(n)` returns the element in `MX` (see Section 3), whose first n upper diagonals agree with the first n upper diagonals of g.

`trall(n)`:
Let n be a positive integer. Suppose we type `trall(n)` in the shell. All the commutators involving elements g_i of M which are then computed will treat each g_i as `gi.trunc(n)`. `trall(0)` resets the file to the original state, that is, operations are computed without any truncation taking place.

Example 4.

```
>>> x0**(-1)
M_0 ([[11, 11, 11], [11, 11, 11], [11, 11, 11], [11, 11, 0], [11, 0, 0]])
>>> g=M(0,[11,11,11])
>>> g**(-1)
M_0 ([[11, 11, 11]],?)
>>> precinv=5
>>> x0**(-1)
M_0 ([[11, 11, 11], [11, 11, 11], [11, 11, 11], [11, 11, 0], [11, 0, 0]])
>>> g**(-1)
M_0 ([[11, 11, 11], [26, 26, 26], [17, 17, 17], [11, 11, 11], [26, 26, 26]],?)
```

Example 5.

```
>>> precinv=5
>>> a=x0.conj(x1)
>>> a
M_0 ([[23, 224, 138], [53, 27, 395], [9, 381, 248], [15, 166, 144], [56, 131, 217],
[18, 192, 79], [26, 69, 1], [14, 1, 2], [1, 2, 3], [2, 3, 0], [3, 0, 0]])
>>> a.trunc(5)
M_0 ([[23, 224, 138], [53, 27, 395], [9, 381, 248], [15, 166, 144], [56, 131, 217]],?)
>>> g=M(0,[11,11,11])
>>> g.conj(x1)
```

```
M_0 ([[23, 224, 138], [53, 27, 395], [16, 426, 82], [17, 104, 128], [52, 128, 11]],?)
>>> (x0.conj(x1))*(x0.comm(x1)) == x1
True
>>> x0.comm(x1,x2)==(x0.comm(x1)).comm(x2)
True
>>> x1.commr(x2,x3)
M_3 ([[28, 235, 129], [26, 26, 26], [0, 157, 106], [23, 224, 11], [46, 144, 473],
[8, 81, 194], [63, 293, 155], [4, 73, 375], [26, 223, 344], [48, 334, 18], [37, 16, 261],
[18, 72, 41], [44, 200, 2], [13, 144, 4], [16, 96, 506], [32, 424, 0], [16, 128, 365],
[0, 64, 292], [40, 64, 146], [32, 128, 292], [16, 256, 365], [32, 320, 0], [40, 0, 0]])
```

Example 6.

```
>>> g=M(0,[11,11,11])
>>> g+x0
M_1 ([[17, 17, 17], [26, 26, 26], [11, 11, 0], [17, 0, 0]])
>>> x0*g
M_1 ([[11, 11, 11], [17, 17, 17], [26, 26, 17], [11, 26, 0], [11, 0, 0]])
>>> g*x0
M_1 ([[11, 11, 11], [17, 17, 17], [26, 26, 17], [11, 0, 26], [0, 0, 11]])
```

3 The class MX

3.1 Instances of MX

An instance g of MX differs from one of M only for the fact that we have information about the first say $k+l$ upper diagonals of g, but we do not know what the other upper diagonals look like.
We define an element g in MX in the following way. Let $V_1,...,V_l$ be 3×9 matrices and k an integer. Then `g=MX(k,V1,...,Vl)` defines the element of MX with k zero upper diagonals followed by l diagonals defined by $V_1,...,V_l$. Similarly to M, the matrices $V_1,...,V_l$ may be replaced by the corresponding 3-tuples of numbers.

`print g`:
Assume V_1 is not the zero matrix. Then the command `print g` gives `M_k([V1,...,Vl],?)` and we can call k by `g.k` and $[V_1,...,V_l]$ by `g.l`.
Assume now that $V_i = 0$ for all $i \leq j$, for some $1 \leq j \leq m$. Then `print g` gives `M_k+j([Uj+1,...,Um],?)`.

`g`:
The difference between `print g` and `g` in MX is analogous to the difference between the same commands in M.

Example 7.

```
>>> g=MX(3,[0,0,0],[1,2,3],[0,0,0])
>>> print g
M_4 ([matrix([[0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 1, 0, 1, 0, 0, 1, 1]]), matrix([[0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0]])],?)
>>> g
M_4 ([[1, 2, 3], [0, 0, 0]],?)
>>> len(g.l)
2
```

```
>>> h=MX(3,[0,0,0],[0,0,0])
>>> print h
M_5 ([],?)
```

3.2 Operations within the class MX

Except for `==` and `!=`, all the other functions listed in 2.2 can also be used with arguments belonging to `MX`. The output of `*`, `**`, `inv()` `conj()`, `comm()`, `+` will in this case be an element in `MX` (except for when we add an element to itself or raise it to the power 0, which gives `Id`).
We add here some comments. In what follows, for an arbitrary element g of this class we will write $k(g)$ for the number of zero upper diagonals and $l(g)$ for the number of known upper diagonals, the first of which is non-zero.

`g*....*g` versus `g**n`:
In `MX`, multiplying an element g by itself, say n times, via the command `*` applied repeatedly may give information about fewer diagonals, in comparison with raising the same element to the power n via the command `g**n`. This is due to the fact that we can determine $2k(g) + l(g) + 1$ upper diagonals of g^2. However, if we were to multiply g by h, where h has the same number of zero diagonals and known diagonals as g, we would be able to determine only the first $k(g) + l(g)$ upper diagonals of gh.
For this reason, when we type `g**n`, the number n is first written as a sum of powers of 2, say $n = 2^{i_1} + ... + 2^{i_m}$. Subsequently, $g^{2^{i_j}}$ is computed for $1 \leq j \leq m$ by squaring i_j times and then multiplication of the factors $g^{2^{i_j}}$ is performed. On the other hand, `__mul__` performs multiplication termwise.

Example 8.

```
>>> x=MX(0,[28,235,129],[29,211,263])
>>> x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x
M_2 ([],?)
>>> x**16
M_15 ([[28, 235, 129], [0, 0, 0]],?)
```

`g**(-1)*h*g` versus `g.conj(h)`:
The command `g**(-1)*h*g` returns an instance of `MX`, say t_1, with $k(t_1)+l(t_1) = \min(k(g)+l(g), k(h)+l(h))$. On the other hand, `g.conj(h)` returns t_2, with $k(t_2) + l(t_2) = k(h) + \min(k(g) + l(g) + 1,\ l(h))$.

Example 9.

```
>>> w=MX(0,[28,235,129])
>>> z=MX(12,[11,11,9],[1,2,3])
>>> w**(-1)*z*w
M_1 ([],?)
>>> w.conj(z)
M_12 ([[11, 11, 9], [15, 3, 100]],?)
```

`g**(-1)*h**(-1)*g*h` versus `g.comm(h)`:
The command `g**(-1)*h**(-1)*g*h` returns t_1 with $k(t_1) + l(t_1) = \min(k(g) + l(g), k(h) + l(h))$.
Suppose $k(g) + l(g) \geq k(h) + l(h)$. Then `g.comm(h)` returns t_2 with $k(t_2) + l(t_2) = k(g) + \min(k(h) + l(h)+1,\ l(g)) + \delta \cdot \min(k(h)+1,\ k(h)+l(h)+1-l(g))$, where $\delta = 0$ if the minimum in the last summand is negative and 1 otherwise.
If $k(g) + l(g) < k(h) + l(h)$, $k(t_2) + l(t_2) = k(h) + \min(k(g) + l(g) + 1,\ l(h)) + \delta \cdot \min(k(g) + 1,\ k(g) + l(g) + 1 - l(h))$.

Example 10. Assume w and z are defined as in Example 9.

```
>>> z**(-1)*w**(-1)*z*w
M_1 ([],?)
>>> z.comm(w)
M_13 ([[14, 1, 103]],?)
```

4 Operations among classes and comparison operators

As well as multiplying, adding, taking conjugates and commutators of instances of the same class, one can perform these operations with one element in `M` and one in `MX`. The outcome will obviously belong to `MX`.

Example 11. The reader may want to compare this example with Example 5.

```
>>> s=MX(0,[11,11,11])
>>> s.conj(x1)
M_0 ([[23, 224, 138], [53, 27, 395]],?)
```

Besides, we can compare two elements of `MX` or one element of `M` and one of `MX` with the operators `>`, `<`, `>=`, `<=`. The output is explained in what follows.

`__gt__()`:
Let g be an instance of `MX` and h an instance of `M` or `MX`. Then `g>h` returns `True` if all the first $k(g)+l(g)$ upper diagonals of g agree with the first $k(g)+l(g)$ upper diagonals of h and, in the case of h in `MX`, $k(g)+l(g) < k(h)+l(h)$.

`__ge__()`:
Let g be an instance of `MX` and h an instance of `M` or `MX`. Then `g>=h` returns `True` if all the first $k(g)+l(g)$ upper diagonals of g agree with the first $k(g)+l(g)$ upper diagonals of h.

`__lt__()`:
Let g be an instance of `M` or `MX` and h an instance of `MX`. Then `g<h` returns `True` if all the first $k(h)+l(h)$ upper diagonals of g agree with the first $k(h)+l(h)$ upper diagonals of h and, in the case of h in `MX`, $k(h)+l(h) < k(g)+l(g)$.

`__le__()`:
Let g be an instance of `M` or `MX` and h an instance of `MX`. Then `g<=h` returns `True` if all the first $k(h)+l(h)$ upper diagonals of h agree with the first $k(h)+l(h)$ upper diagonals of g.

Example 12.

```
>>> s=MX(0,[11,11,11])
>>> s>=x0
True
>>> p=MX(0,[11,11,11],[1,2,3])
>>> p<s
True
>>> r=s
>>> (r>=s) and (s<=r)
True
>>> (r>s) or (r<s)
False
```

References

[1] N. Peyerimhoff and A. Vdovina, "Cayley graph expanders and groups of finite width", *Journal of Pure and Applied Algebra* 215, no. 11 (2011): 2780-8.

[2] N. Barker, N. Boston, N. Peyerimhoff and A. Vdovina, "An Infinite Family of 2-Groups with Mixed Beauville Structures", *International Mathematics Research Notices*, (2014).