mmgroup

Release mmgroup v1.0.0

Martin Seysen

Apr 23, 2024

CONTENTS:

CHAPTER

ONE

THE MMGROUP API REFERENCE

1.1 Introduction

In the area of mathematics known as group theory, the Monster group $\mathbb{M}$ is the largest finite sporadic simple group. It has order

$2^{46} \cdot 3^{20} \cdot 5^{9} \cdot 7^{6} \cdot 11^{2} \cdot 13^{3} \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31 \cdot 41 \cdot 47 \cdot 59 \cdot 71 =$ 808.017.424.794.512.875.886.459.904.961.710.757.005.754.368.000.000.000 .

The Monster group has first been constructed by Griess [Gri82]. That construction has been simplified by Conway [Con85]. For more information about the Monster group we refer to [Wik23].

The **mmgroup** package is a Python implementation of Conway's construction [Con85] of the Monster group $\mathbb{M}$. Its is the first implementation of the Monster group where arbitrary operations in that group can effectively be performed. On the author's PC (Intel i7-8750H at 4 GHz running on 64-bit Windows) the group multiplication in $\mathbb{M}$ takes less than 30 ms. This is more than five orders of magnitude faster than estimated in 2013 in [Wil13].

The Monster group $\mathbb{M}$ has a rational representation ρ of dimension 196884, see [Con85]. In that representation the denominators of the matrix coefficients are powers of two. So reducing these coefficients modulo a small odd prime p leads to a representation ρ_p of $\mathbb{M}$ over the finite field $\mathbb{F}_p$.

The *mmgroup* package uses highly optimized C programs for calculating in such representations ρ_p of the Monster $\mathbb{M}$. The main ingredient for speeding up the computation in $\mathbb{M}$ is the calculation and the analysis of the images of certain vectors in ρ_p that are called 2A axes in [Con85].

In the description of the **mmgroup** package we use the notation in [Sey20], where an explicit generating set of the Monster $\mathbb{M}$ is given. For a mathematical description of the implementation we refer to [Sey22].

1.2 Installation and test

1.2.1 Installation on 64-bit Windows

For installing the *mmgroup* package on a 64-bit Windows system, python 3 must be installed. Then type:

```
pip install mmgroup
pip install pytest
python -m pytest --pyargs mmgroup -m "not slow"
```

The last command tests the installation.

1.2.2 Installation on Linux and macOS with a 64-bit x86 CPU

For installing the *mmgroup* package on a Linux or macOS system, python 3 must be installed. Then type:

```
pip3 install mmgroup
pip3 install pytest
python3 -m pytest --pyargs mmgroup -m "not slow"
```

The last command tests the installation.

1.2.3 Other platforms

For other platforms the package must be compiled from a source distribution. Therefore the *cibuildwheel* tool may be used, see *Building the mmgroup package with cibuildwheel*.

A general description of the build process is given in the *Guide for developers* of the project documentation.

1.3 Basic structures

Conway's construction of the Monster group starts with the extended binary Golay code $\mathcal{C}$, which is a 12-dimensional linear subspace of the 24-dimensional vector space $\mathbb{F}_2^{24}$. The Golay code has Hamming distance 8. Its automorphism group is the Mathieu group M_{24}, which is a simple group operating as a permutation group on 24 points. These points are identified with the basis vectors of $\mathbb{F}_2^{24}$.

Module `mmgroup` contains fast C routines for computing with the Golay code $\mathcal{C}$, its cocode $\mathcal{C}^*$, and the Mathieu group M_{24}. There are Python classes for a more convenient handling of these objects that wrap these C functions; these classes are described in this section.

In this section we also describe Python classes modelling more complicated mathematical structures. One of these structures is the Parker loop $\mathcal{P}$, which is a non-associative loop that can be constructed as a double cover of the Golay code $\mathcal{C}$. We also consider a group $\mathrm{Aut}_{\mathrm{St}}\mathcal{P}$ of automorphisms of $\mathcal{P}$, which has structure $2^{12}.M_{24}$. Here the normal subgroup of $\mathrm{Aut}_{\mathrm{St}}\mathcal{P}$ or order 2^{12} is isomorphic to the Golay cocode $\mathcal{C}^*$.

Another important ingredient of the construction of the Monster is the Leech lattice Λ, which is the densest lattice in dimension 24. We also consider the Leech lattice modulo 2, which we denote by $\Lambda/2\Lambda$, and the automorphism group Co_1 of $\Lambda/2\Lambda$, which is simple.

1.3.1 The Golay code and its cocode

We deal with the Golay code and its cocode.

Let $\tilde{\Omega}$ be the set of integers $\{i \in \mathbb{Z} \mid 0 \leq i < 24\}$ of size 24 and construct the vector space $\mathcal{V} = \mathbb{F}_2^{24}$ as $\prod_{i \in \tilde{\Omega}} \mathbb{F}_2$. In this document elements of $\mathcal{V}$ are called *bit vectors*. We represent bit vectors as instances of class `GcVector`.

We identify the power set of $\tilde{\Omega}$ with the vector space $\mathcal{V} = \mathbb{F}_2^{24}$ by mapping each subset of $\tilde{\Omega}$ to its characteristic function, which is a bit vector in $\mathcal{V}$. So we may talk about union and intersection of bit vectors in a natural way and use the python operators `|` and `&` for these operations as usual. We use the `+` operator for vector addition of bit vectors. Bit vectors are numbered from `0` to `0xffffff`, with bit `i` of that number corresponding to the `i`-th unit vector.

The Golay code $\mathcal{C}$ is a 12-dimensional linear subspace of $\mathcal{V}$ such that a nonzero vector in $\mathcal{C}$ has weight at least 8. This characterizes the Golay code up to permutation. In this package we chose a specific basis of the Golay code that satisfies the requirements in [Sey20], see *The basis of the Golay code and of its cocode* for details. We represent Golay code words as instances of class `GCode`. Golay code words of weight 8 and 12 are called *octads* and *dodecads*, respectively.

We stress that class *GCode* is not a subclass of class *GcVector*. Instances of these two classes are numbered in a different way. Golay code words are numbered from `0` to `0xfff` with bit `i` of that number corresponding to the `i`-th basis vector of the Golay code. See *The basis of the Golay code and of its cocode* for the chosen basis of the Golay code.

The Golay cocode $\mathcal{C}^*$ is the `12`-dimensional quotient space $\mathbb{F}_2^{24}/\mathcal{C}$. For any $\delta \in \mathcal{C}^*$ the weight of δ is the weight of its lightest representative in $\mathbb{F}_2^{24}$. That weight is at most `4`. There is a unique lightest representative it the weight is less than `4`, and there are six mutually disjoint lightest representatives of an element of weight `4`. A partition of $\mathbb{F}_2^{24}$ given by such a set of disjoint representatives is called a *tetrad*. We represent elements of the cocode of the Golay code as instances of class *Cocode*. Cocode elements are numbered from `0` to `0xfff` with bit `i` of that number corresponding to the `i`-th basis vector of the cocode. Our basis of the cocode is the reciprocal basis of the basis of the Golay code, see *The basis of the Golay code and of its cocode* for details.

The *scalar product* of a Golay code word and a cocode element is the parity of the intersection of the Golay code word and any representative of the cocode element. If `g` is an instance of class *GCode* and `d` is an instance of class *Cocode* then `g & d` is an instance of class *Parity*. Here class *Parity* models an integer modulo `2`, which is also considered as the parity of a bit vector. Standard operations, e.g. `(-1)**p`, work as expected for an instance `p` of class *Parity*.

The standard function `len(x)` returns the (minimal) weight `|x|` of an instance `x` of class *GcVector*, *GCode* or *Cocode*. For instances `g1, g2, g3` of class *GCode*, the expression `g1 & g2` is an instance of class *Cocode* and `g1 & g2 & g3` an instance of class *Parity*; these values have a natural interpretations as bitwise intersections. `g1/4` is a shorthand for `Parity(len(g1)/4)`; and `(g1 & g2)/2` is a shorthand for `Parity(len(g1 & g2)/2)`. These two expressions are called the *power map* and the *commutator*, respectively, in [Asc86]; and `g1 & g2 & g3` is called the *associator* in [Asc86].

The numbering of the Golay code words and of the cocode elements is important for indexing some generators of $\mathbb{M}$ and also some basis vectors in the representations ρ_p of $\mathbb{M}$.

class mmgroup.GCode(*value*)

This class models a code word of the Golay code.

The Golay code is a binary `[24, 12, 8]` code. So there are 2^{12} code words, each word is 24 bit long, and the Hamming distance between two different code words is at least 8.

The 2^{12} Golay code words are numbered from `0` to `0xfff`. Bit `i` in that number refers to the coefficient of the `i`-th basis vector of the Golay code, which may be 0 or 1.

Parameters

value (*see table below for legal types*) – This is the value of the Golay code word to be returned.

Returns

A Golay code vector

Return type

an instance of class *GCode*

Raise

- TypeError if `type(value)` is not in the table given below.
- ValueError if `value` cannot be converted to an instance of class *GCode*.

Depending on its type parameter **value** is interpreted as follows:

Table 1: Legal types for constructor of class `GCode`

type	Evaluates to
`int`	Here the code word with number `value` is returned. `0 <= value < 0x1000` must hold.
`list` of `int`	A list of integers `0 <= i < 24` is interpreted as a list of bit positions to be set in a bit vector. The Golay code word corresponding to that bit vector is returned. Up to 3 erroneous bits are corrected.
class *GCode*	A deep copy of parameter `value` is returned.
class *PLoop*	The Parker loop element `value` is converted to a Golay code word (by dropping its sign) and returned.
class *GcVector*	The bit vector `value` of type *GcVector* is converted to a Golay code word and returned. Up to 3 erroneous bits in that vector are corrected.
class *XLeech2*	The Golay code part of the *XLeech2* object is converted to a Golay code word and returned.
`str`	**Create random element depending on the string** `'r'`: Create arbitrary Golay code word

Standard operations

Addition of Golay code words and scalar multiplication (with scalars of type `int` or *Parity*) are done as usual.

The bitwise and operator & means bitwise *and* of two code words of type *GCode*. The result of such an operation on two Golay code words is a Golay cocode word of type *Cocode*.

The bitwise & operator of a Golay code word and a Golay cocode word of type *Cocode* is not well defined as a bit vector, but it has a well-defined parity. So in this case the result of the & operator is a parity object of type *Parity*.

For the bitwise & operator bit vectors of type *GcVector*, see class *GcVector*.

The ~ operator means bitwise complement as usual; it returns the complemented code word, which is also of type *GCode*.

Standard functions

`len(g)` returns the bit weight of the Golay code word `g`, i.e. the number of bits set in the word `g`.

Special functions

Let `g1`, `g2`, and `g3` be Golay code vectors of type *GCode*.

The expression `g1/4` is a shorthand for `Parity(len(g1)/4)`. It returns `len(g1)/4` modulo 2 as a parity object of type *Parity*.

`GcVector(g1 & g2)` returns the bitwise intersection of `g1` and `g2` as a bit vector of type *GcVector*.

`(g1 & g2)/2` is a shorthand for `Parity(len(GcVector(g1 & g2))/2)`. It returns the halved bit weight of `g1 & g2` modulo 2 as a parity object of type *Parity*. This value is also called the *commutator* of `g1` and `g2`.

`g1 & g2 & g3` returns the parity of the bitwise intersection of `g1`, `g2`, and `g3` as a parity object of type *Parity*. This value is also called the *associator* of `g1`, `g2`, and `g3`.

`abs(g1)` is a shorthand for `abs(PLoop(g1))`, see class *PLoop* for details.

property bit_list

Return the ordered list of positions of bits being set

property bits

Return the Golay code word as a 24-bit vector.

The function returns a list with 24 entries equal to 0 or 1.

property gcode

Return the number of the Golay code word.

Same as property `ord`.

property octad

Return number of octad of the Golay code word as an octad.

> Complements of octads are also accepted as octads. We have `0 <= o < 759` for the returned number `o`.

Raise

- ValueError if the Golay code word is not an octad.

property ord

Return the number of the Golay code word.

We have `0 <= i < 0x1000` for the returned number `i`.

split()

Split the Golay code word `g`.

Returns
A triple `(0, eo, v)` with `g = Omega * eo + v`.

Here `Omega` is the Golay code word containing one bits only, `eo` is `0` or `1`, and `v` is a Golay code word of type *GCode* with `0 <= v1.ord < 0x800`.

This method is for compatibility with the corresponding method in class *PLoop*.

split_octad()

Split an octad from the Golay code word `g`.

Returns
A triple `(0, eo, o)` with `g = Omega * eo + o`.

Here `Omega` is the Golay code word containing one bits only, `eo` is `0` or `1`, and `v` is a Golay code word of type *GCode* which is either the zero code word or an octad, i.e. a code word of weight 8.

This method is for compatibility with the corresponding method in class *PLoop*.

Raise

- ValueError if this is not possible.

theta(*g2=None*)

Return cocycle of Golay code words.

The cocycle `theta` maps a pair of Golay code words to an integer modulo 2. It is linear in its second argument, so it may also be considered as a mapping from the Golay code to the cocode of the Golay code.

Parameters
g2 – `None` (default) or another Golay code word of type *GCode*.

Returns

- If `g2` is a code word of type *GCode*, we return the value `g1.theta(g2) = theta(g1, g2)` as a *Parity* object.
- If `g2` is `None` (default), we return the value `g1.theta() = theta(g1)` as a *Cocode* object. Note that `g1.theta(g2) == g1.theta() & g2` .

The importance of the `theta` function comes from the fact that the multiplication of elements of the Parker loop is based on the cocycle. We embed the set of Golay code words into the set of positive Parker loop elements, which are instances of class *PLoop*.

Let `g1` and `g2` be Golay code words of type *GCode*. Then `PLoop(g1)` and `PLoop(g2)` are the corresponding positive Parker loop elements, and `g1.theta(g2)` is an integer modulo `2` of type *Parity*. We have:

`PLoop(g1) * PLoop(g2) == (-1)**g1.theta(g2) * PLoop(g1 + g2)` .

property vector

Return bit vector corresponding to the Golay code word as an int.

We have `0 <= v < 0x1000000` for the returned number `v`. Bit `i` of the number `v` is the `i`-th coordinate of the corresponding bit vector.

class mmgroup.**GcVector**(*value*)

Models a 24-bit vector in the underlying space of the Golay code.

The 2^{24} bit vectors are numbered from `0` to `0xffffff`. Bit `i` in that number refers to the coefficient of the `i`-th basis vector of the vector space, which may be 0 or 1.

Parameters

value (*see table below for legal types*) – This is the value of the bit vector to be returned.

Returns

A bit vector of 24 bit length.

Return type

an instance of class *GcVector*

Raise

- TypeError if `type(value)` is not in the table given above.
- ValueError if `value` cannot be converted to an instance of class *GcVector*.

Depending on its type parameter **value** is interpreted as follows:

Table 2: Legal types for constructor of class `GcVector`

type	Evaluates to
`int`	Here the bit vector with number `value` is returned. `0 <= value < 0x1000000` must hold.
`list of int`	A list of integers `0 <= i < 24` is interpreted as a list of bit positions to be set in a bit vector. That bit vector is returned.
class *GCode*	The bit vector corresponding to the Golay code word `value` is returned.
class *PLoop*	The Parker loop element `value` is converted to a Golay code word (by dropping its sign) and then to a bit vector. That bit vector is returned.
class *GcVector*	A deep copy of parameter `value` is returned.
`str`	**Create random element depending on the string** `'r'`: Create arbitrary bit vector

Standard operations

Addition of bit vectors and scalar multiplication (with scalars of type `int` or *`Parity`*) are done as usual. The sum of a bit vector and a Golay code word (of type *`GCode`*) is a bit vector. The sum of a bit vector and a cocode word (of type *`Cocode`*) is a cocode word of type *`Cocode`*.

The bitwise and operators `&`, `|`, and `~` operate on two bit vectors as expected. If the other operand is a Golay code vector (of type *`GCode`*) then it is converted to a bit vector.

Standard functions

`len(v)` returns the bit weight of the bit vector `v`, i.e. the number of bits set in the vector `v`.

property bit_list

Return the ordered list of positions of bits being set

property bits

Return the Golay code word as a `24`-bit vector.

Returns a list with `24` entries equal to `0` or `1`.

property cocode

Return number of the cocode word corresponding to the vector.

property gcode

Return number of Golay code word corresponding to the bit vector

Raise

- ValueError if the bit vector is not an Golay code word.

property octad

If the bit vector is an octad, return the number of that octad

Complements of octads are also accepted as octads.

Raise

- ValueError if the bit vector is not an octad.

property ord

Return the number of the bit vector.

We have `0 <= i < 0x1000000` for the returned number `i`.

syndrome(*i=None*)

Return syndrome of cocode element as a bit vector.

Parameters
i – `None` (default) or an integer `0 <= i < 24` used to select a syndrome.

Returns
Syndrome of a Golay cocode element as a bit vector of type *`GcVector`*.

Raise

- ValueError if argument `i` is `None` and the syndrome is not unique.

`v.syndrome(i)` is equivalent to `Cocode(v).syndrome(i)`.

syndrome_list(*i=None*)

Return syndrome of cocode element as list of bit positions.

Parameters
i – `None` (default) or an integer `0 <= i < 24` used to select a syndrome.

Returns
Syndrome of a Golay cocode element as a list of at most four bit positions.

Raise

- ValueError if argument `i` is `None` and the syndrome is not unique.

The syndrome of the Golay cocode element is calculated in the same way as in method **syndrome**. But here the result is returned as an ordered list of bit positions corresponding to the bits which are set in the syndrome.

`v.syndrome_list(i)` is equivalent to `Cocode(v).syndrome_list(i)`.

`property vector`

Return the number of the bit vector.

We have `0 <= i < 0x1000000` for the returned number `i`.

Same as property `ord`.

`class mmgroup.Cocode(`*value*`)`

This class models an element of the cocode of the Golay code.

The Golay code is a binary `[24, 12, 8]` code. So its cocode has 2^{12} elements, each of it is a 24 bit long word (modulo the Golay code).

The 2^{12} Golay cocode elements are numbered from `0` to `0xfff`. Bit `i` in that number refers to the coefficient of the `i`-th basis vector of the Golay cocode, which may be 0 or 1. The chosen basis of the cocode is the reciprocal basis of the chosen basis of the Golay code.

Parameters
`value` (*`see table below for legal types`*) – This is the value of the Golay cocode word to be returned.

Returns
A Golay cocode element

Return type
an instance of class *`Cocode`*

Raise

- TypeError if `type(value)` is not in the table given below.

Depending on its type parameter **value** is interpreted as follows:

Table 3: Legal types for constructor of class `Cocode`

type	Evaluates to
`int`	Here the cocode element with number `value` is returned. `0 <= value <= 0x1000` must hold.
`list` of `int`	A list of integers `0 <= i < 24` is interpreted as a list of bit positions to be set in a bit vector. The cocode word corresponding to that bit vector is returned.
class *Cocode*	A deep copy of parameter `value` is returned.
class *XLeech2*	The *cocode* part of the *XLeech2* object is converted to a cocode element and returned.
`str`	**Create random element depending on the string** `'r'`: Create arbitrary cocode element `'e'`: Create an even cocode element `'o'`: Create an odd cocode element

Standard operations

Addition of cocode elements and scalar multiplication (with scalars of type `int` or *Parity*) are done as usual.

`g & c` and `c & g` are legal operations for a Golay code element `g` of type *GCode* and a cocode element `c` of type *Cocode*. The result of this operation is not well defined as a bit vector, but it has a well-defined parity. Thus `g & c` returns a parity object of type *Parity*.

Standard functions

Let `c` be a cocode element of type *Cocode*. `len(c)` returns the bit weight of a shortest representative of the Golay code element `c`.

Special functions

The expression `c % 2` is a shorthand for `Parity(len(c))`. It returns the parity of `c` as a parity object of type *Parity*.

property cocode

Return the number of the cocode element.

We have `0 <= i < 0x1000` for the returned number `i`.

Same as property `ord`.

property ord

Return the number of the cocode element.

We have `0 <= i < 0x1000` for the returned number `i`.

property parity

Return parity of the cocode element as a *Parity* object.

syndrome(*i=None*)

Return syndrome of cocode element as a bit vector.

Parameters
`i` – `None` (default) or an integer `0 <= i < 24` used to select a syndrome.

Returns
Syndrome of a Golay cocode element as a bit vector of type *GcVector*.

Raise

- ValueError if argument `i` is `None` and the syndrome is not unique.

Any Golay cocode element has either a unique shortest representative of bit weight `<= 3` or precisely six shortest representatives of bit weight `4` which form a partition of the underlying set of the code. Such a partition is called a *tetrad*.

In coding theory, a shortest representative of a cocode element is called a *syndrome*.

If the syndrome is unique, the function returns that syndrome. Otherwise it returns the syndrome containing bit at position `i`.

`syndrome_list`(*i=None*)

Return syndrome of cocode element as list of bit positions.

Parameters
`i` – `None` (default) or an integer `0 <= i < 24` used to select a syndrome.

Returns
Syndrome of a Golay cocode element as a list of at most four bit positions.

Raise

- ValueError if argument `i` is `None` and the syndrome is not unique.

The syndrome of the Golay cocode element is calculated in the same way as in method **syndrome**. But here the result is returned as an ordered list of bit positions corresponding to the bits which are set in the syndrome.

`syndromes_llist`()

Return shortest syndromes of cocode element as list of lists.

The function returns the list of all shortest representatives of the cocode element as a list `ll` of lists. Each entry of `ll` corresponds to a shortest representative. Such an entry is an ordered list of bit positions corresponding to the bits which are set in the representative. Output `ll` is sorted in natural order.

class `mmgroup.`**`Parity`**(*value*)

This class models the parity of a bit vector.

As one might expect, the parity of a bit vector may be odd or even, and it is an element of the field `GF(2)` of two elements.

This means that arithmetic operations `+`, `-`, and `*` work as expected on instances of class *`Parity`*, or on instances of this class combined with integers.

The values `1**p` and `(-1)**p` are defined as usual for an object `p` of class *`Parity`*. Similarly, `g**p` is defined if `g` is an element of a group defined in this package and has order `1` or `2`. More precisely, `g` must be an instance of class `AbstractGroupWord` here.

Bitwise operations `&`, `|`, and `^` are illegal on instances of class *`Parity`*.

`Parity(x)` is defined if `x` an integer, an instance of class *`Parity`*, or an instance of any class defined is this package which has a natural parity, e.g. an instance of class *`GcVector`* or *`Cocode`*.

`int(p)` evaluates to `0` or `1` for an instance of class *`Parity`*.

Implementation remarks

If an object `x` has an attribute or a property `parity` then `Parity(x)` means `Parity(x.parity)` and `x + p` means `Parity(x) + p` for any instance `p` of class Parity.

If an object `x` has an attribute or a property `parity_class` then `x * p` and `p * x` mean `Parity(parity_class(x) * int(p))`. If that attribute or a property has value `True` then `x * p` and `p * x` mean `Parity(x * int(P))`.

property ord

Return `1` if the parity is odd and `0` if it is even

1.3.2 The Parker loop

We deal with the Parker loop, which is a non-associative Moufang loop.

The Parker loop is written multiplicatively its elements can be represented as pairs in the set $\mathcal{P} = \mathcal{C} \times \mathbb{F}_2$, see e.g. [Asc86], [Con85]. We also write 1 for the neutral element $(0, 0) \in \mathcal{C} \times \mathbb{F}_2 = \mathcal{P}$ of the Parker loop and `-1` for its negative $(0, 1)$. Let $\Omega = (\tilde{\Omega}, 0)$, where $\tilde{\Omega}$ is the Golay code word `1,...,1`. Then the center $Z(\mathcal{P})$ of $\mathcal{P}$ is $\{1, -1, \Omega, -\Omega\}$. An element (g, s), $g \in \mathcal{C}$ is called *positive* if $s = 0$ and *negative* if $s = 1$.

Multiplication of two Parker loop elements $(g_1, s_1), (g_2, s_2)$ is given by a cocycle $\theta : \mathcal{C} \times \mathcal{C} \to \mathbb{F}_2$ as follows:

$$(g_1, s_1) \cdot (g_2, s_2) = (g_1 + g_2, s_1 + s_2 + \theta(g_1, g_2)).$$

There are several possibilities for the cocycle θ. We choose a cocycle θ satisfying the requirements given by [Sey20], see *The basis of the Golay code and of its cocode* for details.

We represent elements of $\mathcal{P}$ as instances of class *PLoop*. For Golay code word `g` given as an instance of class *GCode*, the value `PLoop(g)` is the positive element `(g,0)` of the Parker loop, and `-PLoop(g)` is the corresponding negative element `(g,1)`. The constant `PLoopOne` is equal to the neutral element `(0,0)` and the constant `PLoopOmega` is equal to the central element `~PLoopOne`. Here the ~ means bitwise complement of the corresponding Golay code element without changing the sign of a Parker loop element. Thus `PLoopOmega` is the positive Parker loop element corresponding to the Golay code word containing `24` one bits. So the center $Z(\mathcal{P})$ of $\mathcal{P}$ is:

```
[PLoopOne, -PLoopOne, PLoopOmega, -PLoopOmega] .
```

Let `g1, g2` be instances of class *GCode*. For positive elements of the Parker loop the multiplication formula given above can be coded as follows:

```
PLoop(g1) * PLoop(g2) == (-1)**g1.theta(g2) * PLoop(g1 + g2) .
```

The elements of the Parker loop are numbered from `0` to `0x1fff`. Here the numbers of the positive Parker loop elements correspond to the numbers of the Golay code words. The numbering system for the Parker loop is also used for indexing certain elements of the monster group $\mathbb{M}$.

A transversal of $Z(\mathcal{P})$ in $\mathcal{P}$ is used for indexing certain basis vectors of the representation ρ of $\mathbb{M}$. Property `ord` of an instance `a` of class *PLoop* returns the number of that element of the Parker loop. For indexing vectors in ρ it is important to know that `(-a).ord = a.ord ^ 0x1000` and `(~a).ord = a.ord ^ 0x800`. Thus the Parker loop elements `a` with `0 <= a.ord < 0x800` are a transversal of $Z(\mathcal{P})$ in $\mathcal{P}$.

class `mmgroup.`**`PLoop`**(*value=0*)

This class models an element of the Parker Loop.

The Parker loop is a non-associative loop with 2^{1+12} elements. Each element can be interpreted as a signed Golay code word, see class *GCode*. Accordingly, class *PLoop* is a subclass of class *GCode*.

The loop operation is written multiplicatively. The neutral element of the loop (corresponding to the positive zero word of the Golay code) is `PLoopOne`, and its negative is `-PLoopOne`. `PLoopOmega` is the (positive) element corresponding to the all-one vector of the Golay code.

The 2^{1+12} Parker loop elements are numbered from `0` to `0x1fff`. Elements `0` to `0xfff` are are positive and corresponding to the Golay code words with the same number. The element with number `0x1000 ^ i` is the negative of the element with number `i`.

Parameters

`value` (*see table below for legal types*) – This is the value of the Parker loop element to be returned.

Returns

A Parker loop element

Return type

an instance of class *PLoop*

Raise

- TypeError if `type(value)` is not in the table given above.
- ValueError if `value` cannot be converted to an instance of class *PLoop*.

Depending on its type parameter **value** is interpreted as follows:

Table 4: Legal types for constructor of class `PLoop`

type	Evaluates to
`int`	Here the code word with number `value` is returned. `0 <= value < 0x2000` must hold. We have `PLoop(0) == PLoopOne`, `PLoop(0x1000) == -PLoopOne`, and `PLoop(0x800) == ~PLoopOne == PLoopOmega`.
`list` of `int`	Such a list is converted to a Golay code word, see class *GCode*, and the corresponding (positive) Parker loop element is returned.
class *GCode*	The corresponding (positive) Parker loop element is returned.
class *PLoop*	A deep copy of parameter `value` is returned.
class *GcVector*	This is converted to a Golay code word, see class *GCode*, and the corresponding (positive) Parker loop element is returned.
class *XLeech2*	The Parker loop part of the *XLeech2* object is returned.
`str`	**Create random element depending on the string** `'r'`: Create arbitrary Parker loop element

Standard operations

Let `a` be a Parker loop element. The multiplication operator `*` implements the non-associative loop operation. Division by a Parker loop element means multiplication by its inverse, and exponentiation means repeated multiplication, with `a**(-1)` the loop inverse of `a`, as usual.

The `~` operator maps the Parker loop element `a` to `a * PLoopOmega`, leaving the sign of `a` unchanged.

Multiplication with the integer `1` or `-1` means the multiplication with the neutral element `PLoopOne` or with its negative `-PLoopOne`, respectively.

If any of the operations `+`, `-` or `&` is applied to a Parker loop element, that element is converted to a Golay code word of type *GCode*, ignoring the sign. This conversion also takes place if a Parker loop element is multiplied or divided by an integer different from `1` or `-1`.

Standard functions

`len(a)` is a shorthand for `len(GCode(a))`. Here function `GCode(a)` converts `a` to a Golay code word of type *GCode*.

`abs(a)` returns the element in the set `{a, -a}` which is positive.

property ord

Return the number of the Parker loop element.

We have `0 <= i < 0x2000` for the returned number `i`.

parity_class

alias of *GCode*

property sign

Return the sign of the Parker loop element.

This is `1` for a positive and `-1` for a negative element.

split()

Split sign and Omega from Parker loop element `a`.

Returns
a triple `(es, eo, v)` with `a = (-1)**e1 * PLoopOmega**eo * v` .

Here `v` is the unique (positive) element of the Parker loop of type *PLoop* with `0 <= v.ord < 0x800` satisfying that equation. `es` and `eo` are `0` or `1`.

split_octad()

Split Parker loop element `a` into central element and octad

Returns
a triple `(es, eo, o)` with `a = (-1)**e1 * PLoopOmega**eo * o`.

Here `o` is either the neutral Parker loop element `PLoopOne` or Parker loop element corresponding to a positive octad. `es` and `eo` are `0` or `1`. An *octad* is a Golay code word (of type *GCode*) of weight `8`.

Raise

- Raise ValueError if this is not possible.

`mmgroup.`**PLoopZ**(*e1=0*, *eo=0*)

Return a specific central element of the Parker loop.

Parameters

- **e1** (*int*) – Sign, the sign of the element will be `(-1)**e1`
- **eo** (*int*) – Exponent for `PLoopOmega`, default is `0`

Returns
The central element `(-1)**e1 * PLoopOmega**eo` of the Parker loop.

Return type
an instance of class *PLoop*.

Here `PLoopOmega` is the positive element of the Parker loop corresponding to the Golay code word `1,...,1`.

`e1` and `eo` may also be parity objects of type *Parity*.

1.3.3 Octads and suboctads

We deal with octads and certain subsets of octads called suboctads.

An *octad* is a Golay code word of weight `8`. The Golay code has `759` octads. A signed octad is a Parker loop element corresponding to an octad or a complement of an octad. Signed octads are used for indexing some unit vectors of the representation ρ of the monster $\mathbb{M}$.

Function `Octad()` returns a signed octad as an instance of class *PLoop*. Arguments of function `Octad()` are interpreted in the same way as arguments of the constructor for class *PLoop*, but the function raises *ValueError* if the result is not a (possibly complemented) signed octad.

We use some lexical order for numbering the `759` octads, which we do not describe in detail. Due to the well-known properties of the Golay code we can create a random octad and display its number as follows:

```
from random import sample
from mmgroup import Octad
# Select 5 random integers between 0 and 23
int_list = sample(range(24), 5)
# Complete these 5 integers to a (unique) octad o
o = Octad(int_list)
# Display octad o and its number
print("Octad", o.bit_list, "has number", o.octad)
```

A *suboctad* of an octad `o` is an element of the Golay cocode $\mathcal{C}^*$ of even parity which can be represented as a subset of `o`. Each octad has `64` suboctads. A pair (octad, suboctad) is used for indexing some basis vectors in the representation ρ. function `SubOctad` creates a suboctad as an instance of class *XLeech2* from such a pair. Function `SubOctad` takes two arguments `octad` and `suboctad`. The first argument evaluates to a signed octad as in function `Octad()`. The second argument evaluates to a suboctad, see function `SubOctad` for details.

The raison d'etre of a suboctad is indexing a basis vector in the representation ρ. For this purpose we need a pair of integers refering to the octad and the suboctad. For an instance `so` of class *XLeech2* that pair is given as the pair of the last two integers in `so.vector_tuple()`.

For an octad `o` and a suboctad `s` precisely one of the pairs `(o,s)` and `(~o,s)` is actually used as an index for ρ. The pair `(~o,s)` is used if the cocode element corresponding to `s` has minimum weight `2`; the pair `(o,s)` is used if that element has minimum weight `0` or `4`. See [Con85] or [Sey20] for background. In this implementation both, `(o,s)` and `(~o,s)`, refer to the same basis vector of ρ.

`mmgroup.Octad`(*octad*)

Return a (signed) octad as an element of the Parker loop.

Parameters
octad (*see table below for legal types*) – the value of the octad to be returned.

Returns
a (signed) octad

Return type
an instance of class *PLoop*

Raise

- TypeError if `type(octad)` is not in the table given above.
- ValueError if `octad` cannot be converted to a (possibly negated and complemented) octad.

Depending on its type parameter **octad** is interpreted as follows:

Table 5: Legal types for parameter `octad`

type	Evaluates to
`int`	Here the (positive) octad with number `octad` is returned. There are 759 octads in the Golay code. So `0 <= octad < 759` must hold.
`list` of `int`	Such a list is converted to a Golay code word, see class *GCode*, and the corresponding (positive) Parker loop element is returned.
class *GCode*	The corresponding (positive) Parker loop element is returned.
class *PLoop*	A deep copy of parameter `octad` is returned.
class *GcVector*	This is converted to a Golay code word, see class *GCode*, and the corresponding (positive) Parker loop element is returned.
class *XLeech2*	The *octad* part of the vector `octad` is returned.
`str`	**Create random element depending on the string** `'r'`: Create arbitrary octad

A complement of an octad is also accepted; then the corresponding Parker loop element is retured. The function raises ValueError if parameter `octad` does not evaluate to an octad or a complement of an octad.

`mmgroup.octad_entries(`*octad*`)`

Return list of entries of an octad

Here parameter `octad` describes an octad as in function `Octad`. That octad is a sum of eight basis vectors of GF_2^{24}, with basis vectors numbered from 0 to 23. The function returns the list `[b0,...,b7]` of the indices of these eight basis vectors in a certain order that may differ from the natural order.

We use these indices for numbering the suboctads of that octad as described in the documentation of function `SubOctad`.

Caution: The order of the basis vectors returned by this function may be changed in future versions!

`mmgroup.SubOctad(`*octad*, *suboctad=0*`)`

Creates a suboctad as instance of class *XLeech2*

Parameters

- **octad** (same as in function *Octad*) – The first component of the pair *(octad, suboctad)* to be created.
- **suboctad** (*see table below for legal types*) – The second component of the pair *(octad, suboctad)* to be created.

Returns

The suboctad given by the pair *(octad, suboctad)*

Return type

an instance of class *XLeech2*

Raise

- TypeError if one of the arguments *octad* or *suboctad* is not of correct type.
- ValueError if argument *octad* or *suboctad* does not evaluate to an octad or to a correct suboctad, respectively.

A *suboctad* is an even cocode element that can be represented as a subset of the octad given by the argument *octad*.

The raison d'etre of function `SubOctad` is that pairs *(octad, suboctad)* are used for indexing vectors in the representation of the monster group. Here we want to number the octads from `0` to `758` and the suboctads form `0` to `63`, depending on the octad. Note that every octad has `64` suboctads.

Depending on its type parameter **suboctad** is interpreted as follows:

Table 6: Legal types for parameter `suboctad`

type	Evaluates to
`int`	Here the suboctad with the number given in the argument is taken. That numbering depends on the octad given in the argument `octad`. `0 <= suboctad < 64` must hold.
`list` of `int`	Such a list is converted to a bit vector as in class *GcVector*, and the cocode element corresponding to that bit vector is taken.
class *GCode*	The intersection of the octad given as the first argument and the Golay code word given as the second argument is taken.
class *GcVector*	This is converted to a cocode element, see class *Cocode*, and that cocode element is taken.
class *Cocode*	That cocode element is taken as the suboctad.
`str`	**Create random element depending on the string** `'r'`: Create arbitrary suboctad

The numbering of the suboctads

Suboctads are numbered for 0 to 63. Let `[b0, b1,..., b7]` be the bits set in the octad of the pair `(octad, suboctad)`. Here we assume the indices of the basis vectors making up the octad are ordered as returned by function `octad_entries`, when applied to the octad.

The following table shows the suboctad numbers for some suboctads given as cocode elements. More suboctad numbers can be obtained by combining suboctads and their corresponding numbers with `XOR`.

Table 7: Suboctad numbers of some cocode elements

`[b0,b1]`	`[b0,b2]`	`[b0,b3]`	`[b0,b4]`	`[b0,b5]`	`[b0,b6]`
`s = 1`	`s = 2`	`s = 4`	`s = 8`	`s = 16`	`s = 32`

E.g. `[b0, b5, b6, b7]` is equivalent to `[b1, b2, b3, b4]` modulo the Golay code and has number `s = 1 ^ 2 ^ 4 ^ 8 = 15`.

1.3.4 Automorphisms of the Parker loop

We deal with a certain group of automorphisms of the Parker loop.

A *standard automorphism* is an automorphism of the Parker loop $\mathcal{P}$ which maps to an automorphism of the Golay code $\mathcal{C}$ when factoring out the elements $\{1, -1\}$ of $\mathcal{P}$. Let $\mathrm{Aut}_{\mathrm{St}}\mathcal{P}$ be the group of standard automorphisms of $\mathcal{P}$. We embed $\mathrm{Aut}_{\mathrm{St}}\mathcal{P}$ into $\mathbb{M}$ as in [Con85]. In ATLAS notation, see [CCN+85], that group has structure

$$\mathrm{Aut}_{\mathrm{St}}\mathcal{P} = 2^{12}.M_{24} \,.$$

This means that $\mathrm{Aut}_{\mathrm{St}}\mathcal{P}$ has an elementary Abelian normal subgroup of order 2^{12} with factor group M_{24}. That group extension does not split. Here the group 2^{12} is isomorphic to the Golay cocode $\mathcal{C}^*$ and M_{24} is the Mathieu group acting on $\mathbb{F}_2^{24}$ by permuting the basis vectors. M_{24} is the automorphism group of the Golay code $\mathcal{C}$.

The automorphisms in the subgroup $\mathcal{C}^*$ of $\mathrm{Aut}_{\mathrm{St}}\mathcal{P}$ are called *diagonal* automorphisms. A diagonal automorphism $d \in \mathcal{C}^*$ maps an element a of $\mathcal{P}$ to $(-1)^s \cdot a$ with $s = |a \,\&\, d|$. Here $\&$ means the bitwise `and` operation of bit vectors and $|.|$ means the bit weight of a bit vector.

Since the extension $2^{12}.M_{24}$ does not split, there is no canonical embedding of M_{24} into $\mathrm{Aut}_{\mathrm{St}}\mathcal{P}$. For practical purposes we need such an embedding. We choose a basis $b_1, \ldots, b_{11}$ of the Golay code $\mathcal{C}$, see section *The basis of the Golay code and of its cocode*. Let a_i be the positive element of the Parker loop $\mathcal{P}$ corresponding to b_i. For any $\tilde{g} \in M_{24}$ we let g be the unique element of $\mathrm{Aut}_{\mathrm{St}}\mathcal{P}$ with $g\mathcal{C}^* = \tilde{g}$ that maps all elements $a_1, \ldots, a_{11}$ of $\mathcal{P}$ to positive elements of $\mathcal{P}$. We call g the *standard representative* of $\tilde{g}$ in $\mathrm{Aut}_{\mathrm{St}}\mathcal{P}$.

Elements of $\mathrm{Aut}_{\mathrm{St}}\mathcal{P}$ are modelled as instances of class `AutPL`. If `d` is an instance of class `Cocode` representing an element of $\mathcal{C}^*$ then `AutPL(d)` is the corresponding diagonal automorphism.

A permutation $p \in M_{24}$ can be represented as a list `l_p = [p(0),...,p(23)]`. If `l_p` is such a list then `AutPL(l_p)` is the standard representative of `p` in $\mathrm{Aut}_{\mathrm{St}}\mathcal{P}$. Here an error occurs if the mapping from `i` to `p(i)` is not in M_{24}. A permutation in M_{24} can also be given as a mapping from a subset of $\{0, \ldots, 23\}$ to $\{0, \ldots, 23\}$, provided that this mapping extends to a unique element of M_{24}. For details, see class `AutPL`.

The group $\mathrm{Aut}_{\mathrm{St}}\mathcal{P}$ operates naturally on $\mathcal{P}$ by right multiplication. It also operates on $\mathbb{F}_2^{24}$, $\mathcal{C}$, and $\mathcal{C}^*$ by right multiplication. Here the basis vectors of $\mathbb{F}_2^{24}$ are permuted by M_{24}; so the kernel of that operation is the subgroup $\mathcal{C}^*$ of $\mathrm{Aut}_{\mathrm{St}}\mathcal{P}$.

We order the elements p of M_{24} in lexicographic order based on the lists `l_p = [p(0),...,p(23)]`. We number the `244823040` elements of M_{24} in that order from `0` to `244823039`. Thus the number `0` is assigned to the neutral element of M_{24}, and the number `244823039` is assigned to the permutation `p` with `l_p` =

$$[23, 22, 21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 0, 1, 2, 3, 4, 5, 6, 7]$$

These numbers are used for addressing the corresponding standard representatives in the subgroup $\mathrm{Aut}_{\mathrm{St}}\mathcal{P}$ of the monster $\mathbb{M}$.

One way to create an element of M_{24} is to map an *umbral heptad* to another umbral heptad. Here an umbral heptad is a set of `7` elements of $\{0, \ldots, 23\}$ such that precisely `6` of these elements are contained in an octad, see [CS99], Chapter 10. Then the `6` elements in one octad must map to the `6` elements in the other octad.

The set `[0,1,2,3,4,5,8]` is an umbral heptad with `8` not contained in the octad. We may create the standard representative (in $\mathrm{Aut}_{\mathrm{St}}\mathcal{P}$) of a random element of M_{24} as follows:

```python
from random import sample
from mmgroup import Octad, AutPL
# Create a random octad
o = Octad("r")
# Select 6 random elements from that octad
hexad = sample(o.bit_list, 6)
# Select one random element not in that octad
extra = sample((~o).bit_list, 1)
# Create umbral heptad from these random elements
heptad = hexad + extra
# Create a mapping from [0,1,2,3,4,5,8] to that heptad
hmap = zip([0,1,2,3,4,5,8], heptad)
# Create a permutation in the Mathieu group from that mapping
aut = AutPL(0, hmap)
# 'aut' is the standard representative of the permutation
# Show 'aut' as a permutation in form of a list
```

(continues on next page)

(continued from previous page)

```
print(aut.perm)
# Show the automorphism 'aut' with its permutation number
print(aut)
```

Remark

Class *AutPL* is a subclass of class `mmgroup.structures.AbstractGroupWord` which implements elements of an abstract group and the operations on such elements. For an instance `g` of *AutPL*, the object `g.group` is an instance of a subclass of `mmgroup.structures.AbstractGroup`. That subclass models the group $\mathrm{Aut}_{\mathrm{St}}\mathcal{P}$.

Generating random elements of certain subgroups of M_{24}

We support the generation of random elements of certain subgroups of M_{24}. Here any such subgroup stabilizes a subset (or a sets of subsets) of the set $\tilde{\Omega} = \{0, \ldots, 23\}$ on which M_{24} acts. Furthermore, any such group is named by a flag (which is a character), as indicated in the table below. Intersections of these subgroups are described by combining the characters. The character `r` describes the whole group M_{24}. A string of characters describing such a subgroup should start with `r`. Whitespace is ignored.

Table of subgroups:

Flag	Mnemonic	Subgroup stabilizing the set	Structure
'r'	(random)	the whole set $\tilde{\Omega}$	M_{24}
'2'	2-set	$\{2, 3\}$	$M_{22} : 2$
'o'	octad	$\{0, \ldots, 7\}$	$2^4 : A_8$
't'	trio	$\{\{8i, \ldots, 8i + 7\} \mid i < 3\}$	$2^6 : (S_3 \times L_3(2))$
's'	sextet	$\{\{4i, \ldots, 4i + 3\} \mid i < 6\}$	$2^6 : 3.S_6$
'l'	(line)	$\{\{2i, 2i + 1\} \mid 4 \leq i < 12\}$	$2^{1+6} : L_3(2)$
'3'	3-set	$\{1, 2, 3\}$	$L_3(4) : S_3$

For mathematical background we refer to section *Subgroups of the Mathieu group M_{24}*.

Table 8: Examples of strings describing subgoups of M_{24}

String	Action
`'r'`	Generate a random element of M_{24}
`'r 2o'`	Generate a random element of M_{24} stabilizing $\{0, \ldots, 7\}$ and $\{2, 3\}$

class `mmgroup.`**`AutPL`**(*d=0*, *p=0*, *unique=1*)

This class models a standard automorphism of the Parker loop.

Parameters

- **d** – This parameter describes an element of the Golay cocode. If `d` is an instance of class *AutPL* then this describes an automporphism of the Parker loop; in that case parameter `p` must be set to its default value. Legal types of parameter `d` see below. The parameter defaults to zero word of the cocode.
- **p** – This parameter describes a permutation in the Mathieu group `M_24` or, more precisely, the standard representative of that permutation in the automporphism group of the Parker loop. Legal types of that parameter see below. The parameter defaults to neutral element of `Mat24`.
- **unique** – If this is `True` (default) and parameter `p` is not `r` then parameter `p` must describe a unique permutation in the Mathieu group `M_24`. Otherwise we take the least possible permutation (in lexicographic order) that agrees with parameter `p`.

Returns
A standard Parker loop automorphism

Return type
an instance of class *AutPL*

Raise

- TypeError if `type(value)` is not in the table given below.
- ValueError if the set of arguments cannot be converted to an instance of class *AutPL*.

Table 9: Legal types for parameter `d` in constructor of class `AutPL`

type	Evaluates to
`int`	The number of an element of the Golay cocode.
class *Cocode*	This represents an element of the Golay cocode.
`str`	**Create random element depending on `str`** `'r'`: Create an arbitrary cocode element `'e'`: Create an even cocode element `'o'`: Create an odd cocode element Any other string is illegal.
class *AutPL*	A deep copy of the given automorphism in *AutPL* is returned. Then parmeter `p` must be set to its default value.

Table 10: Legal types for parameter `p` in constructor of class `AutPL`

type	Evaluates to	
`int`	Here the integer is the number of a permutation in the Mathieu group `M_24`.	
`list` of `int`	A list `l_p` of `24` of disjoint integers `0 <= i < 24` is interpreted as a permutation in `M_24` that maps `i` to `l_p[i]`.	
`dict`	A dictionary specifies a mapping from a subset of the integers `0 <= i < 24` to integers `0 <= dict[i] < 24`. This mapping must extend to a permutation in `M_24`. If parameter `unique` is `True` (default) then that permutation must be unique in `M_24`.	
`zip` object	`zip(x,y)` is equivalent to `dict(zip(x,y))`.	
`str`	Create random element depending on the string	
	`'r'`	Create random element of `M_24`
	`'r <flags>'`	Create random element of subgroup of `M_24`, depending on `<flags>`, see explanation above.

Let `a` be an instance of class *GcVector*, *GCode*, *Cocode*, or *PLoop*, and let `g1` , `g2` be instances of class *AutPL*. Then `a * g1` is the result of the natural operation of `g1` on `a`, which belongs to the same class as `a`.

`g1 * g2` means group multiplication, and `g1 ** n` means exponentiation of `g1` with the integer `n`. `g1 ** (-1)` is the inverse of `g`. `g1 / g2` means `g1 * g2 ** (-1)`.

`g1 ** g2` means `g2**(-1) * g1 * g2`.

`as_tuples`()

Convert group element to a list of tuples

For a group element `g` the following should hold:

`g.group.word(*g.as_tuples()) == g` .

So passing the tuples in the list returned by this method as arguments to `g.group` or to `g.group.word` reconstructs the element `g`.

This shows how to convert a group element to a list of tuples and vice versa.

check()

Check automorphism for consistency via 'assert' statements

`a.check()` returns `a`.

property cocode

Return number of cocode element in the automorphism.

For an instance `g` of *AutPL* we have

`g == AutPL(Cocode(g.cocode)) * AutPL(g.perm)`.

property parity

Return parity of an automorphism

This is `s` if `PLoopOmega` maps to `(-1)**s * PLoopOmega` for `s == 0` or `s == 1`.

property perm

Return permutation of automorphism as a list `p_l`.

Then the permutation maps `i` to `p_l[i]` for `i = 0,...,23`.

property perm_num

Return the number of the permutation of the automorphism

The Mathieu group `M_24` has `244823040` elements numbered from `0` to `244823039` in lexicographic order. Thus the neutral element has number `0`.

1.3.5 The Leech lattice modulo 2 and its preimage Q_{x0}

We deal with an extraspecial group that maps to the Leech lattice modulo 2

Let Q_{x0} be the group generated by elements x_d, x_δ with $d \in \mathcal{P}, \delta \in \mathcal{C}^*$. Here $\mathcal{P}$ is the Parker loop and $\mathcal{C}^*$ is the Golay cocode as in [Con85]. We take the following relations in Q_x from [Sey20]:

$$x_d x_e = x_{d \cdot e} x_{A(d,e)},\ x_\delta x_\epsilon = x_{\delta\epsilon},\ [x_d x_\delta] = x_{-1}^{\langle d,\delta\rangle};\ d, e \in \mathcal{P};\ \delta, \epsilon \in \mathcal{C}^* .$$

Here $A(d, e)$ is the associator between d and e, i.e. the element of $\mathcal{C}^*$ corresponding to the vector $d \cap e$; and $[.,.]$ is the commutator in Q_{x0}. Then Q_{x0} is an extraspecal group of structure 2_+^{1+24}.

An element of the group Q_x is modelled as an instance of class *XLeech2*.

In that class we encode the element $x_d \cdot x_\delta$ of Q_{x0} as an integer x as follows:

$$x = 2^{12} \cdot d \oplus (\delta \oplus \theta(d)) .$$

Here elements of the Parker loop and elements of the cocode are encoded as integers as in section *The Parker loop* and *The Golay code and its cocode*. θ is the cocycle given in section *The basis of the Golay code and of its cocode*, and '$\oplus$' means bitwise addition modulo 2. Note that a Parker loop element is 13 bits long (with the most significant bit denoting the sign) and that a cocode element is 12 bits long.

There is a natural homomorphism from Q_{x0} to the Leech lattice Λ modulo 2 with kernel $\{x_1, x_{-1}\}$, as described in [Con85]. This homomorphism maps the group operation in Q_{x0} to vector addition in $\Lambda/2\Lambda$. With our numbering in Q_{x0} that homomorphism can be realized by simply dropping the sign bit 25; then vector addition in $\Lambda/2\Lambda$ corresponds to the $\oplus$ operation on such numbers. The natural quadratic form on the vector in $\Lambda/2\Lambda$ with the number $2^{12} \cdot d + \delta$, $0 \leq d, \delta < 2^{12}$ is the parity of the bit vector $d \oplus \delta$.

class mmgroup.**XLeech2**(*ploop=0*, *cocode=0*, **args*)

This class models an element of the group Q_{x0}.

The group Q_{x0} is an extraspecial 2 group of structure 2^{1+24}.

The group operation is written multiplicatively.

The 2^{1+24} group elements are numbered from `0` to `0x1ffffff`. Elements `0` to `0xffffff` are considered positive. The element with number `0x1000000 ^ i` is the negative of the element with number `i`.

An element is constructed as a product $x_d x_\delta$, where d is in the Parker loop and δ is in the Golay cocode.

Parameters

- **value** – This parameter describes the value of an element of the group Q_{x0} as indicated in the table below.
- **cocode** – This parameter describes an element x_δ (with δ in the Golay cocode) to be right multiplied to the element of Q_{x0} obtained from the first parameter.

Returns

An element of the group Q_{x0}.

Return type

an instance of class *XLeech2*

Raise

- ValueError if the input cannot converted to an element of the group Q_{x0}.
- TypeError the ype of an input is illegal.

Caution: Although the group Q_{x0} has a natural embedding into the subgroup G_{x0} of the monster group, we do not consider an instance q of class *XLeech2* as an element of G_{x0}. We consider q as a (possibly negated) basis vector of a space on which the subgroup G_{x0} of the monster acts monomially by right multiplication.

Depending on its type parameter **value** is interpreted as follows:

Table 11: Legal types for constructor of class `XLeech2`

type of `value`	Evaluates to
`int`	Here the code word with number `value` is returned. `0 <= value < 0x2000000` must hold.
class *XLeech2*	A deep copy of the object `value` is created.
class *GCode*	The corresponding Golay code element is converted to a (positive) element of Q_{x0}.
class *PLoop*	The corresponding Parker loop element is converted to an element of Q_{x0}.
class *MM*	Then `value` in an element of the monster group. If that element is in the subgroup Q_{x0} of the monster, then it is converted to the coresponding instance of class *XLeech2*.
`'r'`	Create random element depending on the string, see explanation below.
A letter `BCTXE`	If `value` is a single capital letter in `BCTXE` then we create an element corresponding to a unit vector in ρ_p as described below.
Any other string `s`	This is intepreted as the element `MM('q', s)` of the Monster group.

If value is of one of the type listed above then the following parameter is converted to an element of the Golay cocode (as in the constructor of class *Cocode* multiplied to the converted parameter `value` from the right.

Alternatively, parameter `value` may be a single letter. If `value` is one of the capital letters `BCTXE` then that letter and the following arguments are passed to the constructor of class *MMVector* in order to generate a (possible negated) basis vector of the representation ρ_p of the monster. (Here the characteristic p is ignored.) If that basis vector corresponds to an element of Q_{x0} then we construct that element of Q_{x0}; otherwise an error occurs.

If parameter `value` is the letter `'r'` then we generate a random element of Q_{x0}. If an integer `i` is given as a second parameter `'r'` then we generate a random element of Q_{x0} of type `i`. Here the type of a vector $v \in \Lambda/2\Lambda$ is the halved norm of the shortest vector in the set $v + 2\Lambda$. If `i` is specified then it must be equal to 0, 2, 3 or 4.

Standard operations

Let `q` be an instance of this class. The multiplication operator `*` implements the group operation. Division by an element means multiplication by its inverse, and exponentiation means repeated multiplication, with `q**(-1)` the inverse of `q`, as usual.

Multiplication with the integer `1` or `-1` means the multiplication with the neutral element or with the central involution x_{-1}.

The opration `&` denotes the scalar product of the vectors in the Leech lattice modulo 2 obtained from an instance of this class, ignoring the sign.

Standard functions

`abs(a)` returns the element in the set `{a, -a}` which is positive.

If an instance q of class *XLeech2* maps to a vector of type 2 in the Leech lattice modulo 2 then q is also a unit vector in the 196884-dimensional representation ρ of the monster group.

One may use q in the constructor of class *MM* (representing the monster group) for creating the corresponding element of the subgroup Q_{x0} of the monster group.

as_Leech2_bitvector()

Convert element to a bit vector in a numpy array

The element is returned as a numpy array `v` of length 24 of 8-bit unsigned integers, with each entry equal to 0 or 1. This array represents a vector in the Leech lattice mod 2 in the standard encoding.

For an instance `x` of this class we have `v[i] = (x.ord >> i) & 1`, for 0 <= `i` < 24.

as_suboctad(*strict=False*)

Convert element to a suboctad if possible.

Let $x_d \cdot x_\delta$ be equal to the given element of Q_{x0}. If d corresponds to a (possibly complemented) octad and δ is an even cocode element and can be represented as a subset of d then the given element is a suboctad. In this case the function returns a pair *(octad, suboctad)* of integers describing a suboctad a in function `SubOctad` in Section *Octads and suboctads*.

If `strict` is True then the given element must also correspond to a short vector in the Leech lattice mod 2. The function raise ValueError if the given element does not satisfy the conditions mentioned above.

classmethod gen_type(*vtype=2*, *positive=True*)

Return generator that yields the vectors of a given type

The function returns a generator that yields all elements of Q_{x0} that map to a vector of the type `vtype` in the Leech lattice (mod 2) as instances of class *XLeech2*.

Parameter `vtype` must be 0, 2, 3 or 4, where `vtype = 2` (default) means the short Leech lattice vectors.

If parameter `positive``is ``True` (default) then the elements with positive sign are generated only.

isplit()

Split element into a product $x_d \cdot x_\delta$

The method returns a pair (d, δ) such that $x_d \cdot x_\delta$ is equal to the given element of Q_{x0}.

Here (d, δ) is returned as a pair of integers.

property mmdata

Return internal representation of corresponding monster element.

That internal representation is returned as a numpy array of 32-bit integers.

octad_number()

Return number of octad.

Let $x_d \cdot x_\delta$ be equal to the given element of Q_{x0}. If d corresponds to a (possibly complemented) octad in the Golay code then the function returns the number of that octad as a nonnegative integer less than 759. Otherwise the function raises `ValueError`.

Here the octads are numbered as in class *GCode*.

property ord

Return the number of the element of Q_{x0} .

We have `0 <= i < 0x2000000` for the returned number `i`.

property sign

Return the sign of the element Q_{x0}.

This is `1` for a positive and `-1` for a negative element.

split()

Split element into a product $x_d \cdot x_\delta$

The method returns a pair (d, δ) such that $x_d \cdot x_\delta$ is equal to the given element of Q_{x0}.

The element d of the Parker loop is an instance of class *PLoop*. The element δ of the Golay cocode is an instance of class *Cocode*.

str()

Represent group element as a string

property subtype

Return pair `(type, subtype)` of the element of Q_{x0}

Here `type` is as in property `type`, and subtype is a one-digit decimal integer describing the subtype of a vector in the Leech lattice modulo 2, as explained in the **guide** in section *Computations in the Leech lattice modulo 2*.

property type

Return the type of the element of Q_{x0}.

This is equal to the type of the corresponding vector v in the Leech lattice Λ modulo 2. The type of a vector $v \in \Lambda/2\Lambda$ is the halved norm of the shortest vector in the set $v + 2\Lambda$. That type is equal to 0, 2, 3 or 4.

vector_tuple()

Return unit vector in representation of the monster as tuple

If the element of Q_{x0} corresponds to a vector of type 2 in the Leech lattice then it also corresponds to a unit vector `v` in the representation ρ of $\mathbb{M}$; otherwise the function raises ValueError.

The function returns a tuple `(sign, tag, i0, i1)`, with `sign = +1 or -1`. Then the triple `(tag, i0, i1)` corresponds to a basis vector `u` in ρ as described in class *MMVector*, and we have `v = sign * u`.

property xsubtype

Return subtype of the element of Q_{x0}

The function returns the integer `16 * type + subtype`, where the pair `(type, subtype)` is as in property `subtype`. See section *Computations in the Leech lattice modulo 2* in the **guide** for background.

`mmgroup.`**`leech2_orbits_raw`**(*g_list*)

Compute orbits of the Conway group on the Leech lattice mod 2

The Conway group Co_1 has a natural action on $\Lambda/2\Lambda$, i.e. on the Leech lattice mod 2. The subgroup G_{x0} of the Monster (of structure $2^{1+24}.\mathrm{Co}_1$) has the same action on $\Lambda/2\Lambda$.

Given a list `g_list` of generators of a subgroup H of G_{x0}, the function computes the orbits of $\Lambda/2\Lambda$ under the action of H. Here the entrries of the list `g_list` may be instances of class *MM*, which are elements of the group G_{x0}.

The function encodes an element of $\Lambda/2\Lambda$ as an integer, as in class *XLeech2*, setting the sign bit to zero.

The function returns a triple `(n_sets, indices, data)` that defines the partition of $\Lambda/2\Lambda$. Here the integer `n_sets` is the number of sets in the partition. `indices` is an array of indices referring the the array `data`. Array `data` stores the 2^{24} elements of $\Lambda/2\Lambda$, so that for `0 <= i < n_sets` the `i`-th set of the partition is equal to the array `data[indices[i] : indices[i+1]]`.

The entries of the array `data[indices[i] : indices[i+1]]` are sorted. The sets of the partition are sorted by their smallest elements.

1.3.6 The basis of the Golay code and of its cocode

We chose a basis `b_0`,…, `b_11` of the Golay code $\mathcal{C}$ such that the resulting Golay code is compatible to [CS99], Chapter 11.

As usual in the Python or C language we number the indices of the basis vectors of the space $\mathbb{F}_2^{24}$ from `0` to `23`. Similarly, we number the basis vectors of the Golay code from `0` to `11`.

In [CS99] the Golay code is described in terms of the MOG (Miracle Octad Generator), which can be considered as a 4×6 matrix for arranging the basis vectors of $\mathbb{F}_2^{24}$. Our numbering of these basis vectors corresponds to the MOG as follows:

0	4	8	12	16	20
1	5	9	13	17	21
2	6	10	14	18	22
3	7	11	15	19	23

For the cocode $\mathcal{C}^*$ we use the reciprocal basis `c_0`,…, `c_11` of the basis `b_0`,…, `b_11` of $\mathcal{C}$. Then the scalar product `b_i & c_j` of `b_i` and `c_j` is equal to `1` if `i == j` and to `0` otherwise.

The basis vectors `b_0`,…, `b_11` of the Golay code are given in the following table. The table also shows the cocyles of the basis vectors as explained below:

```
      Basis vectors of the Golay code            Cocycle theta
      binary: bit 0,...,23             hex       bit 0,...,11
b_0:  0000 1111 0000 1111 1111 1111,  0xfff0f0;  0111 0000 0000
b_1:  0000 1111 1111 0000 1111 1111,  0xff0ff0;  1011 0000 0000
```

(continues on next page)

(continued from previous page)

```
 b_2:  0000 1111 1111 1111 0000 1111,  0xf0fff0;   1101 0000 0000
 b_3:  0000 1111 1111 1111 1111 0000,  0x0ffff0;   1110 0000 0000
 b_4:  0000 0000 0011 0011 0011 0011,  0xcccc00;   1111 0000 0000
 b_5:  0000 0000 0101 0101 0101 0101,  0xaaaa00;   1111 0000 0000
 b_6:  0000 0011 0000 0011 0101 0110,  0x6ac0c0;   0111 0000 0000
 b_7:  0000 0101 0000 0101 0110 0011,  0xc6a0a0;   0111 0000 0000
 b_8:  0011 0000 0000 0011 0110 0101,  0xa6c00c;   1000 0000 0010
 b_9:  0101 0000 0000 0101 0011 0110,  0x6ca00a;   1000 0000 0010
b_10:  0111 1000 1000 1000 1000 1000,  0x11111e;   0000 0000 0000
b_11:  1111 1111 1111 1111 1111 1111,  0xffffff;   0000 0000 0000
```

Representatives of the basis vectors `c_0`,. . . , `c_11` of the Golay cocode are given in the following table:

```
       Basis vectors of the cocode
       binary: bit 0,...,23          hex
 c_0:  0000 1000 1000 0000 0000 0000,  0x000110
 c_1:  0000 1000 0000 1000 0000 0000,  0x001010
 c_2:  0000 1000 0000 0000 1000 0000,  0x010010
 c_3:  0000 1000 0000 0000 0000 1000,  0x100010
 c_4:  0000 0000 0101 0000 0000 0000,  0x000a00
 c_5:  0000 0000 0011 0000 0000 0000,  0x000c00
 c_6:  0000 0101 0000 0000 0000 0000,  0x0000a0
 c_7:  0000 0011 0000 0000 0000 0000,  0x0000c0
 c_8:  0101 0000 0000 0000 0000 0000,  0x00000a
 c_9:  0011 0000 0000 0000 0000 0000,  0x00000c
c_10:  1000 1000 1000 1000 1000 1000,  0x111111
c_11:  1000 0000 0000 0000 0000 0000,  0x000001
```

The cocycle $\theta : \mathcal{C} \times \mathcal{C} \rightarrow \mathbb{F}_2$ is used for the multiplication of two elements of the Parker loop, see section *The Parker loop*. For basis vectors b_i, b_j of $\mathcal{C}$ the column `theta` in the table above contains the value $\theta(b_i, b_j)$ in bit j of row i. From these values other values of the cocycle θ can be computed by using

$$\theta(g_1 + g_2, g_3) = \theta(g_1, g_3) + \theta(g_2, g_3) + |g_1 \& g_2 \& g_3|,$$
$$\theta(g_1, g_2 + g_3) = \theta(g_1, g_2) + \theta(g_1, g_3), \quad g_1, g_2, g_3 \in \mathcal{C} \; .$$

Here $\&$ means the bitwise `and` operation of bit vectors and $|.|$ means the bit weight of a bit vector.

The cocycle θ has been chosen in accordance with the requirements in [Sey20]. In that context the *grey* subspace of $\mathcal{C}$ is spanned by $b_0, \ldots, b_3, b_{10}, b_{11}$ and the *coloured* subspace of $\mathcal{C}$ is spanned by $b_4, \ldots, b_9$. Similarly, the *grey* subspace of $\mathcal{C}^*$ is spanned by $c_0, \ldots, c_3, c_{10}, c_{11}$ and the *coloured* subspace of $\mathcal{C}^*$ is spanned by $c_4, \ldots, c_9$.

These specific properties of our chosen basis and cocycle are vital for obtaining an effective implementation of a complete generating set of the monster $\mathbb{M}$.

1.4 The Monster group

We deal with the representation of elements of the monster group.

1.4.1 Generators of the monster group $\mathbb{M}$

Conway [Con85] has constructed a 196884-dimensional rational representation ρ of the monster $\mathbb{M}$ based on representations of two subgroups $G_{x0} = 2_+^{1+24}.\mathrm{Co}_1$ and $N_0 = 2^{2+11+22}.(M_{24} \times S_3)$ of $\mathbb{M}$ which generate $\mathbb{M}$. Here G_{x0} has a normal extraspecial 2-subgroup 2_+^{1+24} with factor group Co_1, where Co_1 is the automorphism group of the Leech lattice modulo 2. The group N_0 has a normal subgroup $2^{2+11+22}$, which is a certain 2 group and the factor group is a direct product of the Mathieu group M_{24} and the symmetric permutation group S_3 of 3 elements.

The group $N_{x0} = N_0 \cap G_{x0}$ has index 3 in N_0 and structure $2^{1+24}.2^{11}.M_{24}$. It is generated by elements x_δ, x_π, x_e, y_e, z_e, for all $x_\delta \in \mathcal{C}^*$, $\pi \in \mathrm{Aut}_{\mathrm{St}}\mathcal{P}$ and $e \in \mathcal{P}$.

Here $\mathcal{C}^*$ is the Golay cocode defined in section *The Golay code and its cocode*, $\mathcal{P}$ is the Parker loop defined in section *The Parker loop*, and $\mathrm{Aut}_{\mathrm{St}}\mathcal{P}$ is the automorphism group of $\mathcal{P}$ defined in section *Automorphisms of the Parker loop*. The group N_0 has a subgroup isomorphic to $\mathrm{Aut}_{\mathrm{St}}\mathcal{P}$ generated by the generators $x_\delta, \delta \in \mathcal{C}^*$ and $x_\pi, \pi \in \mathrm{Aut}_{\mathrm{St}}\mathcal{P}$. The generators x_δ generate the subgroup of diagonal automorphisms of $\mathrm{Aut}_{\mathrm{St}}\mathcal{P}$.

The group N_0 is generated by N_{x0} and by another element τ of N_0 or order 3. The group G_{x0} is generated by N_{x0} and by another element ξ of G_{x0} or order 3. The elements x_δ, x_π, x_e, y_e, z_e, τ and ξ generate the monster group $\mathbb{M}$. In this package we use the definitions of the generators in [Sey20], which incorporate a modification of [Con85] made in [Iva09]. This leads to simpler relations in N_0. The generators x_e, y_e, and z_e in [Sey20] correspond to the generators $x_e \cdot z_{-1}^{|e/4|}$, $y_e \cdot x_{-1}^{|e/4|}$, and $z_e \cdot y_{-1}^{|e/4|}$ in [Con85].

1.4.2 Representing elements of the monster group

An element of the monster group $\mathbb{M}$ is represented as an instance of class `MM` in module `mmgroup`.

Elements of the monster group are created by the constructor of class *MM*. Usually, the constructor of class *MM* takes two arguments `tag` and `i`, where `tag` is a single small letter describing the type of the generator of $\mathbb{M}$, and `i` is an integer describing the value of that generator. Alternatively, `i` may be an instance of the appropriate algebraic structure used for indexing a generator of a given type as indicated in the table below.

1.4.3 Implementation of the generators of the monster group

Math papers may use (at least) Latin or Greek letters for labelling objects, but most programming languages are restricted to ASCII characters. The following table shows how to create generators of the monster group using the constructor of class *MM*:

Table 12: Construction of generating elements of the monster

Element	Construction as an instance of class `MM`,
x_δ	`MM('d', delta)`, `delta` an instance of class *Cocode*; `MM('d', delta)` returns `MM('d', Cocode(delta))` for `0 <= delta < 0x1000`.
x_π	`MM(pi)`, `pi` an instance of class *AutPL*; `MM('d', delta) * MM('p', n)` is equivalent to `MM(AutPL(delta, n))` for `0 <= delta < 0x1000`, `0 <= n < 244823040` .
x_e	`MM('x', e)`, `x` an instance of class *PLoop*; `MM(('x', e))` returns `MM('x', PLoop(e))` for `0 <= e < 0x2000`.
y_e	`MM('y', e)`, `e` as in case x_e.
z_e	`MM('z', e)`, `e` as in case x_e.
τ^e	`MM('t', e)`, exponent `e` is an integer which is taken modulo 3.
ξ^e	`MM('l', e)`, exponent `e` is an integer which is taken modulo 3.

More possibilities for constructing elements of an instance of class *MM* are given in the description of that class. Multiplication and exponentiation of group elements works a usual.

1.4.4 Multiplication of elements of the monster group

Internally, an element of $\mathbb{M}$ is represented as a word of the generators given above. Multiplication with the `*` operator is a concatenation of such words, followed by a reduction step. Multiplication of two reduced elements of the monster group (including the reduction of the result) takes less than 50 milliseconds on the author's computer.

The reduction after a group operation is done by a new method that tracks pairs of perpendicular 2A axes, see [Sey22] for details.

1.4.5 Python classes implementing the Monster group

class `mmgroup.MM`(*tag=None*, *i=None*, **args*, ***kwds*)

Models an element of the monster group $\mathbb{M}$

Let `g1` and `g2` be instances of class *MM* representing elements of the monster group. Then `g1 * g2` means group multiplication, and `g1 ** n` means exponentiation of `g1` with the integer `n`. `g1 ** (-1)` is the inverse of `g`. `g1 / g2` means `g1 * g2 ** (-1)`. We have `1 * g1 == g1 * 1 == g1` and `1 / g1 == g1 ** (-1)`.

`g1 ** g2` means `g2**(-1) * g1 * g2`.

Let `V` be a vector space that is a representation of `MM`, see class *MMVector* for details. An instance `g1` of class `MM` operates on the vector space `V` by right multiplication.

Variables

- **tag** – In the simplest case, parameter `tag` is a string of length `1` that determines the type of the atomic group element. There are also some special cases for parameter `tag `` as indicated below. If ``tag` is not given or equal to `1` then the neutral element of the monster group is constructed.
- **i** – Parameter `i` is number of the atomic element of a given `tag`. Depending on the tag, `i` may be the number of an element of one of the structures *PLoop*, *Cocode*, or the number of an element of the Mathieu group `M_24`, as explained in class *AutPL*. An element π of the group *AutPL* is mapped to the element x_π of the Monster group.

  The number `i` may also be an instance of the appropriate class *PLoop*, *Cocode*, or *AutPL*, as indicated in the table below.

Returns
An element of the monster group

Return type
an instance of class *MM*

Standard tags

The tags listed in the following tables are standard tags that can be used for creating generators (or some short words of generators) of the monster group.

Table 13: Atomic elements of the Monster group

Tag	Number i	Type of element
'p'	0-244823039	The automorphism `AutPL(0, i)` of the Parker loop, see below.
'd'	0-0xfff	The diagonal automorphism `Cocode(i)` in *AutPL*.
'x'	0-0x1fff	The element x_e, with `e = PLoop(i)`.
'y'	0-0x1fff	The element y_e, `e = PLoop(i)`; similar to tag `'x'`.
'z'	0-0x1fff	The element z_e, `e = PLoop(i)`; similar to tag `'x'`. We have $x_e y_e z_e = y_e x_e z_e = 1$.
't'	0-2	The element τ^i,
'l'	0-2	The element ξ^i,
'q'	0-0x1ffffff	Describes an element of the subgroup Q_{x0}. See remark below.
'c'	0-0xffffff	A representative of a right coset of N_{x0} in G_{x0}. See remark below.

Apart from the standard tags there are also some tags for special purposes discussed in the following sections.

Remarks

If `i` is the string `'r'` then a random element with the given tag is generated. If `i` is the string `'n'` then a random element with the given tag is generated, which is different from the neutral element with a very high probability.

If `tag == 'd'` then `i = 'o'` generates a random odd and `i = 'e'` generates a random even diagonal automorphism. In this case i` may also be an instance of class *Cocode*, representing the diagonal automorphism corresponding to the given element of the Golay cocode.

If `tag == 'p'` then `i` may also be an instance of class *AutPL*. Then the returned atom is the Parker loop automorphism given by that instance. If `i` is an integer then the Parker loop automorphism `AutPL(0, i)` is returned. This automorphism is the standard representative of the i-th element of the Mathieu group `M_24` in the automorphism group of the Parker loop.

If `tag == 'p'` then `i` may also be a string starting with the letter `r`. This string indicates a certain subgroup of the Mathieu group `M_24` as described in section *Generating random elements of certain subgroups of M_{24}*. Then we generate a random element `g` of that subgroup, and the returned atom is the standard representative `AutPL(0, g)` of `g` in the automorphism group of the Parker loop.

If `tag` is `'x'`, `'y'` or `'z'` then `i` may also be an instance of class *PLoop*, representing an element of the Parker loop.

The exponent `i` for a tag `'t'` or `'l'` is reduced modulo `3`.

The tag `'q'` is useful for encoding an element of the subgroup Q_{x0} of the monster. An atom with tag `'q'` and index `i` encodes the element $x_d \cdot x_\delta \cdot x_{\theta(d)}$, with d = `PLoop(i >> 12)` $\in \mathcal{P}$‘, δ = `Cocode(i & 0xfff)` $\in \mathcal{C}^*$, and $\theta : \mathcal{P} \rightarrow \mathcal{C}^*$ the cocycle of the Parker loop.

The tag `'c'` with index `i` encodes a representative of the right coset $N_{x0}g$, $g \in G_{x0}$ of N_{x0} in G_{x0} that maps the standard frame Ω in the Leech lattice modulo 2 to a type-4 vector given by the index `i`. Here `i` encodes a vector in the Leech lattice modulo 2 as in the description for tag `'q'`. That vector must be of type 4. The sign bit of that vector is ignored. Here `i` may also be an instance of class *XLeech2* representing a type-4 vector. We warn the user that the index `i` may specify any element of the corresponding right coset $N_{x0}g$; and that this element may depend on the version of the package.

Generating a random element of the subgroup of the monster

If parameter `tag` is equal to the string `'r'` then parameter `i` should be a string describing a subgroup of the monster group. E.g. if parameter `i` is the string `'G_x0'` then a uniform distributed element of the subgoup `'G_x0'` of the Monster is generated. The group `'G_x0'` is the centralizer of the involution x_{-1}; and it has structure $2^{1+24}.\mathrm{Co}_1$. For more information about generating random elements in subgroups of the Monster we refer to Section *Generating random elements of certain subgroups of the Monster* in the **API reference**.

Here Parameter `i` may also be a positive integer. If this is the case then an element of the monster of shape $g_0 t_1 g_1 t_2 g_2 ... t_i g_i$ is created, where g_j is a random element of `'G_x0'`, and t_j is $\tau^{\pm 1}$.

Generating an element of monster from its internal representation

If parameter `tag` is equal to the string `'a'` then parameter `'i'` should be an array-like object containing the internal representation of an element of the monster group.

Internally, an element of the monster group is represented as an array of unsigned 32-bit integers, where each entry of the array describes a generator. See section *Header file mmgroup_generators.h* for details.

Special types accepted as tags

In the simplest case the argument `tag` in the constructor is a string of length `1`. There are a few other types which are also accepted as tags. For such a special tag, the following parameter `i` in the constructor is ignored.

Especially, we may encode a word of standard generators as a list of tuples as indicated in the table below.

Table 14: Legal types as tags in the constructor of class `MM`

Object of type	Evaluates to
type `list`	A list represents the product of its entries in the given order. An entry `MM(tag, i)` can be abbreviated to the tuple `(tag, i)`.
class *PLoop*	The Parker loop element d given by that object is mapped to $x_d \in \mathbb{M}$.
class *AutPL*	The Parker loop automorphism π given by that object is mapped to $x_\pi \in \mathbb{M}$.
class *Cocode*	The Golay cocode element δ given by that object is mapped to $x_\delta \in \mathbb{M}$.
class *XLeech2*	The element corresponding to that element of Q_{x0} is created.
class *MM*	A deep copy of the given element of $\mathbb{M}$ is made.
`str` of length greater than `1`	For an instance `g` of class `MM` we have `MM(str(g)) == g`. So we can reread printed elements of `MM`.

Special strings accepted as a value after tag 'q'

The following strings are accepted after tag 'q'. They denote special elements in the group Q_{x0} as indicated in the following table.

String after tag `q`	Evaluates to
`'+', '-'`	x_1, x_{-1}
`'Omega', '-Omega'`	$x_\Omega, x_{-\Omega}$, where Ω is the Golay code word $(1, \ldots, 1)$
`'omega', '-omega'`	$x_\omega, x_{-1} x_\omega$, for the cocode word ω = [0,1,2,3]
`'v+', 'v-'`	$x_\delta, x_{-1} x_\delta$, for the cocode word δ = [2,3], see [Sey22] background

The strings `'+', '-', 'Omega', '-Omega'` are also accepted after a tag `x`, meaning the same element. They are also accepted after tags `x, z` with the obvious meaning, e.g. `('y', '-Omega')` = $y_{-\Omega}$, `('z', '-')` = z_{-1}.

`as_Co1_bitmatrix()`

Convert element of G_{x0} to a 24 times 24 bit matrix

The bit matrix is returned as a 24 times 24 bit numpy array of 8-bit unsigned integers, with each entry equal to 0 or 1.

That matrix operates by right multiplication on a vector representing an element of the Leech lattice mod 2. See method `mmgroup.XLeech2.as_Leech2_bitvector` for the encoding of such a vector.

The function raises ValueError if the element is not in the subgroup G_{x0} of the Monster.

`as_int()`

Map element of the Monster to a 255-bit integer

The method returns a 255-bit integer `n` describing the element of the Monster group. This integer is unique for each element of the Monster, so that it can be used e.g. for storing large subsets of the Monster in a hash table.

Function `MM_from_int` reconstructs the element of the Monster from the number returned by this function.

Warning:

The numbering used here is not contiguous and not portable between difffernt versions of the `mmgroup` package!

For writing a element to a file, you should convert it to a string instead!

`as_tuples()`

Convert group element to a list of tuples

For a group element `g` the following should hold:

`self.__class__(self.as_tuples()) == self` .

This shows how to convert a group element to a list of tuples and vice versa.

`chi_G_x0(`*involution=None*`)`

Compute characters of element of subgroup G_{x0}

If the element is in the subgroup G_{x0} then the function returns a tuple $(\chi_M, \chi_{299}, \chi_{24}, \chi_{4096})$ of integers. Otherwise it raises `ValueError`.

Here χ_M is the character of the element in the 196833-dimensional rep 198883_x of the monster.

By Conway's construction of the monster we have:

$198883_x = 299_x \oplus 24_x \otimes 4096_x \oplus 98280_x$,

for suitable irreducible representations $299_x, 24_x, 4096_x, 98280_x$ of the group G_{x0}. The corresponding characters of the element of G_{x0} are returned in the tuple given above.

While the product $\chi_{24} \cdot \chi_{4096}$ is well defined, the factors χ_{24} and χ_{4096} are defined up to sign only. We normalize these factors such that the first nonzero value of the pair (χ_{24}, χ_{4096}) is positive.

If parameter `involution` is a 2B involution i then the element g given by `self` must centralize that involution i. In this case we compute the corresponding characters of g^h for a $h \in \mathbb{M}$ such that i^h is the central involution of G_{x0}.

`chi_powers(`*max_e=None*, *ntrials=100*, *mp=1*`)`

Return order and some character information of the element

Let g be the element of the Monster given by `self`; and let χ_M be the character given by the 196833-dimensional rep 198883_x of the monster. The method tries to compute characters $\chi_M(g^e)$ for values e dividing the order of g. This method is very fast; but it may fail, as discusssed below.

The function returns a triple `(o, chi, h)`, where `o` is the order of the element g.

Short description:

Part `chi` of the return value is a dictionary that maps the divisors `e` of the order `o` to the character values $\chi_M(g^e)$. We put `chi[e] = None` if $\chi_M(g^e)$ has not been computed.

If `o` is even then $\chi_M(g^{o/2})$ is computed with a very high probability. So we can check if g powers up to a 2A or to a 2B involution. If g powers up to a a 2B involution then all characters are computed with high pobability.

Exact description:

The method succeeds with high probability if g powers up to a 2B involution, and also if $g \in G_{x0}$. In other cases the method usually fails. If the character of an element is computed then the characters of the powers of that element are also computed.

The method tries to find an element h of the Monster such that computing the character of $h^{-1}gh$ is easier than computing the character of g. Here $h = 1$ is possible. The function returns h in part `h` of the return value. We use a probabilistic algorithm to compute h.

If o is even and `chi[o//2]` is not `None` then we asssert that $h^{-1}g^{o/2}h$ is equal to the standard 2A or 2B involution. The standard 2A involution is x_δ, where δ is the Golay cocode element $(2, 3)$. The standard 2B involution is the central involution x_{-1} of G_{x0}. Here the computation of `chi[o//2]` may fail with a very small probability, depending on the number of trials, as given by parameter `ntrials`.

If parameter `max_e` is an integer (i.e. not `None`), we may omit some computations of `chi[e]` for `e > max_e`. We always try to compute `chi[o//2]` if `o` is even.

If parameter `mp` is greater than 1 then we may use up to `mp` parallel processes.

conjugate_involution(*check=True*, *ntrials=40*, *verbose=0*)

Find an element conjugating an involution into the centre

If the element g given by `self` is an involution in the monster then the method computes an element h of the monster with $h^{-1}gh = z$, where z is defined as follows:

If $g = 1$, we put $h = z = 1$

if g is a 2A involution (in the monster) then we let z be the involution in Q_{x0} corresponding to the Golay cocode word with entries $2, 3$ being set.

if g is a 2B involution (in the monster) then we let z be the central involution in G_{x0}

The function returns a pair `(I, h)`, where h as an element of the instance `MM` of class `MMGroup`. We put `I = 0` if $g = 1$. We put `I = 1, 2` if g is a 2A or 2B involution, respectively.

This function may take a long time. Parameter `ntrials` gives the number of trials to find a suitable element h. Default is `ntrials = 40`. The function may fail after that number of trials even if the element is a 2B involution.

If parameter `check` is True (default) then the function first checks if `g` is an involution.

In future versions support for multiprocessing may be added.

conjugate_involution_G_x0(*guide=0*, *group=None*)

Wrapper for corresponding method in class `mmgroup.Xsp2_Co1`

Here `self` must be an involution g in the subgroup G_{x0} of the Monster. This function performs the same computation as the corresponding method in class `mmgroup.Xsp2_Co1`. It returns a pair `(iclass, a)` such that $h = a^{-1}ga$ is a (fixed) representative of the class of g in the group G_{x0}. Here `h` is described by the integer `iclass` as documented in the corresponding method mentioned above.

If `group` is `None` (default) then `a` is an instance of the same class as `g`.

copy()

Return a deep copy of the group element

half_order(*max_order=119*)

Return the (halved) order of the element of the monster group

The function returns a pair `(o, h)`, where `o` is the order of the element.

If `o` is even then the function returns the pair `(o, h)`, where `h` is the element raised to to the `(o/2)`-th power. Otherwise the function returns the pair `(o, None)`.

Parameter `max_order` is as in function `check_mm_order`.

If `h` is in the subgroup G_{x0}‘ then `h` is returned as a word in the generators of that subgroup.

half_order_chi(*ntrials=40*)

Return order and some character information of the element

This method is deprecated. use method `chi_powers` instead!

The function returns a triple `(o, chi, h)`, where `o` is the order of the element g of the monster given by `self`.

If the order o of g is odd then the triple `(o, None, None)` is returned.

If the order of g is even then we return the triple `(o, chi, h)`. Here `h` is an element h of $\mathbb{M}$ such that $z = h^{-1} g^{o/2} h$ is the standard 2A or 2B involution as in method `conjugate_involution`.

If g powers up to a 2B involution then `chi` is equal to the character of $u = h^{-1} g h$. Here the character of the element u of G_{x0} is returned as a quadruple as in method `chi_G_x0`.

If g powers up to a 2A involution then `chi` is equal to to `None`.

in_G_x0()

Check if the element is in the subgroup G_{x0}

The function returns True if this is the case. If this is the case then the element is converted to a word in the generators of G_{x0}.

This method uses geometric information about the Leech lattice taken from [Iva99].

in_N_x0()

Check if the element is in the subgroup N_{x0}

The function returns True if this is the case. If this is the case then the element is converted to a word in the generators of N_{x0}.

in_Q_x0()

Check if the element is in the subgroup Q_{x0}

The function returns True if this is the case. If this is the case then the element is converted to a word in the generators of Q_{x0}.

is_reduced()

Return `True` if the element of the monster group is reduced

An element `g` of the monster group represented by an instance of this class may be reduced by calling `g.reduce()`.

property mmdata

Return the internal representation of the group element

Internally, an element of the monster group is represented as an array of unsigned 32-bit integers, where each entry of the array describes a generator. See section *Header file mmgroup_generators.h* for details.

order(*max_order=119*)

Return the order of the element of the monster group

We use the method in [LPWW98], section 7, for computing the order of an element of the monster.

If the argument `max_order` is present then the order of the element is checked up to (and including) `max_order` only. Then the function returns `0` if the order is greater than `max_order`. By default, the function returns the exact order of the element.

reduce()

Reduce a group element

This method reduces the group element in place. The reduced form of an element of the Monster is unique; i.e. it depends on the value of the element only, and not on its representation as a word of generators.

The following methods of this class reduce an instance of this class: `as_int()`. Converting an instance of this class to a string, by using method `__str__()` or the built-in function `str()`, also performs a reduction prior to the conversion.

Other methods designed for internal purposes, e.g. `as_tuples()` or `mmdata`, may or may not reduce an instance of this class. The result of a group operation may or may not be reduced.

Note that the reduction of an element of the Monster is a time-comsuming operation, so that we'd better do it in a lazy way.

Finding an *optimal* or *shortest* representation of an element as a word of generators is beyond our current capabilties. Also, the reduction process depends on many internal details. Thus a future version of this package may return a different *reduced* form of an element of the Monster.

1.4.6 Functions dealing with elements of the Monster group

`mmgroup.`**`MM_from_int`**`(`*n*`)`

Obtain an element of the Monster from its number `n`

if `g` is an instance of class `MM` then we have

`MM_from_int(g.as_int()) == g`

1.4.7 Generating random elements of certain subgroups of the Monster

In this section we present various subgroups of the Monster known by the `mmgroup` package. Each of these groups is labelled ba a string. In the documentation of class *`MM`* we have already seen the basic way how to generate a random element of a subgroup of the Monster. E.g. calling method `MM('r', 'G_x0')` generates an element of the subgroup G_{x0} of structure $2^{1+24}.\mathrm{Co}_1$. We may use some other names instead of `'G_x0'` for generating random elements in other subgroups of the Monster. Details will be given in the following subsections.

At present, these names are used to generate random elements of the corresponding subgroup. In future versions of `mmgroup` we may also deal with constructive membership testing in these groups, and with intersections of these groups. So we will only label groups, where the embedding of this group in the Monster is sufficiently well understood.

Describing a random subgroup of the Monster

The subgroups of the Monster recognized by the `mmgroup` package fall in two (not necessarily disjoint) categories. There are small subgroups that are given by the sets of their generators; and there are large 2-local subgroups which are given by their centralizers. Each subgroup has one or more names; and it is described in the following two sections.

The intersection of two groups is obtained by joining their names with an `'&'` character. E.g, `'G_x0 & N_0'` means the intersection of the two groups with names `'G_x0'` and `'N_0'`. We have chosen the definition of the subgroups carefully, so that at least in principle their intersections can be computed.

But not all possible computations have been implemented. For each group there is a column 'Supported' in the following tables, with an entry 'full', 'yes', or 'no'. If all groups in a tuple are supported 'full' then we may compute intersections of these groups. If a group has an entry 'yes' then we may use that group. Otherwise using that group might lead to an error.

A function dealing with the name of a subgroup may raise `NotImplementedError` if that subgroup is not supported.

Caution:

The user should check the corresponding entry 'Supported' in the tables in the follwing two subsections before using the name of a group! Using a name of a group that is not supported might fail without notice!

Some small subgroups of the Monster

The following small subgroups of the Monster are recognized:

Table 15: Some small subgroups of the Monster

Name	Structure	Generators	Supported
N_0	$2^{2+11+22}.(S_3 \times M_{24})$	$x_d, y_d, x_\delta, x_\pi, \tau$	full
N_0_e	$2^{2+11+22}.(3 \times M_{24})$	$x_d, y_d, x_\delta, x_\pi, \tau, \delta$ even	full
N_x0	$2^{1+24+11}.M_{24}$	$x_d, y_d, x_\delta, x_\pi$; equal to $G_{x0} \cap N_0$	full
N_x0_e	$2^{2+11+22}.M_{24}$	$x_d, y_d, x_\delta, x_\pi, \delta$ even	full
Q_x0	2^{1+24}	x_d, x_δ	no
AutPL	$2^{11}.M_{24}$	x_δ, x_π	no

Notation for the generators is as in *mm_group*.

Large 2-local subgroups of the Monster

We have implemented (or will implement) the following 2-local subgroups of the Monster:

Table 16: Large 2-local subgroups of the Monster

Name	Structure	Normalizer of group generated by	Supported
G_x0, G_1	$2^{1+24}.\mathrm{Co}_1$	x_{-1}	full
N_0, G_2	$2^{2+11+22}.(S_3 \times M_{24})$	$x_{-1}, x_{\pm\Omega}$	full
G_3	$2^{3+6+12+18}.(L_3(2) \times 3S_6)$	$x_{-1}, x_{\pm\Omega}, x_\omega$	no
G_5t	$2^{5+10+20}.(S_3 \times L_5(2))$	$x_{-1}, x_{\pm\Omega}, x_\omega, x_{\{0,1,4,5\}}, x_{\{0,2,4,6\}}$	no
G_5l	?	$x_{-1}, x_{\pm\Omega}, x_\omega, x_{\{0,1,4,5\}}, x_{\{0,2,4,7\}}$	no
G_10	$2^{10+16}.\Omega^+_{10}(2)$	$x_d, y_d, x_{ij}, \quad 0 \leq i < j < 8,$ $d \in \{-1, \pm\Omega, d_8\}$	no
B	$2.B$	$x_{\{2,3\}}$	yes
2E_6	$2^2.{}^2E_6(2) : S_3$	$x_{\{2,3\}}, x_{\{2,4\}}, x_{\{3,4\}}$	no

Notation is as in the previous section. Furthermore, d_8 is the standard octad $\{0, 1, 2, 3, 4, 5, 6, 7\}$. ω is the element of the Golay cocode given by the tetrad $\{0, 1, 2, 3\}$ as in [Sey20].

The names *G_x0* and *N_0* are adopted from [Con85]. The names and structures of *G_1, G_2, G_3, G_5t, G_5l* and *G_10* are taken from [Iva09], Chapter 4.1 et seq. We have *G_1 = G_x0* and *G_2 = N_0*. The other names indicate the large simple subgroups involved in that group. Note that all these subgroups are maximal in the Monster, except for *G_5l*.

1.4.8 Long-term stable storage of elements of the Monster group

The *mmgroup* package has been used in several research projects. In some of these projects, elements of the Monster group have been stored permanently in a file or published in a research paper. Here it is important to understand how elements of the Monster should be stored permanently, and how they should be reread into an application using a different version of the *mmgroup* package.

Short recommendation:

Always convert elements of the Monster group to strings before storing them permanently or publishing them! Always convert such strings to instances of class `MM` before using them!

Detailed explanation:

For permanent storage, an element of the Monster given as an instance of class `MM` should be converted to a string by using one of the standard python functions `str` or `print`. We will ensure that future versions of the *mmgroup* package can interpret such a string correctly. Here Release 0.0.8 of the *mmgroup* package is interoperable with all later releases. The standard way to convert such a string back to an instance of class `MM` is to apply the constructor of that class to that string.

Comparing elements of the Monster by comparing strings is not feasible. Here the strings should be converted to instances of class `MM` before comparing them. One reason for this restriction is that at present we do not know how to find a shortest possible reduced form of an element. So future versions of *mmgroup* may implement better reduction algorithms.

Method `as_int` of class `MM` converts an element of the Monster group to an integer. It is ok to compare two such integers when they have been generated by the same version of the *mmgroup* package. When switching to a new version, such integers should be converted to instances of class `MM` using function `MM_from_int` in the *mmgroup* package; and then they should be converted to strings. Pickling instances of class `MM` with the standard python module `pickle` is also discouraged, when switching to another version of the *mmgroup* package.

1.5 The representation of the Monster group

We deal with the 196884-dimensional representation of the monster.

The monster $\mathbb{M}$ has a 196884-dimensional representation ρ with coefficients in $\mathbb{Z}[\frac{1}{2}]$, see [Con85]. Representation ρ is called 196884_x in [Con85]. We obtain a modular representation ρ_p of $\mathbb{M}$ by reducing the matrix coefficients of the representation ρ modulo an odd number p. The representation ρ_p can be implemented very efficiently if $p+1$ is a small power of two. The current version of the *mmgroup* package supports the representations ρ_p of $\mathbb{M}$ for `p = 3, 7, 15, 31, 127, 255`. In the sequel we are a bit sloppy, calling ρ_p a vector space also in cases `p = 15, 255`. The user may select these values for obtaining representations modulo `5` and `17`, respectively.

One purpose of the `mmgroup` package is to give the user the capability to compute in the `196884`-dimensional representation ρ_p of the monster group $\mathbb{M}$.

For calculating in ρ_p the user may create an instance of class `MM` that models an element `g` of the monster group $\mathbb{M}$ as described in section *The Monster group*. Then the user may create an instance `v` of class `MMVector` that models a vector in ρ_p, with `p` as above. The element `g` of $\mathbb{M}$ acts on the vector `v` by right multiplication.

1.5.1 Creating basis vectors of the representation space ρ_p

By construction of ρ_p it is natural to label a basis vector of ρ_p by a tuple `(tag, i0, i1)` with integers `i0, i1`, as described in the next section. Here `tag` is a single capital letter and indices `i0`, `i1` are unsigned integers; these three quantities describe a basis vector of the space `V`.

To create a basis vector of ρ_p corresponding to the tuple `(tag, i0, i1)` the user may call the constructor `MMVector(p, tag, i0, i1)`.

For a fixed characteristic `p` the function `MMV(p)` returns an object corresponding to the vector space ρ_p. Thus the sequence

```
>>> from mmgroup import MMV
>>> V = MMV(p)
>>> v = V(tag, i0, i1)
```

creates the same basis vector `v` as the statement `v = MMVector(p, tag, i0, i1)`.

1.5.2 Description of the basis vectors of the representation space

The following table shows the tags available for describing basis vectors. The column `Vector` in that table shows the basis vector in the notation of [Sey20]. The column `Tuple` shows the basis vector as a tuple `(tag, i0, i1)`.

Table 17: Basis vectors of the representation of the monster

Vector	Tuple	Remarks
$(ij)_1$	`('A',i,j)`	`0 <= i,j < 24`; we have $(ij)_1 = (ji)_1$.
X_{ij}	`('B',i,j)`	`0 <= i,j < 24, i != j`; we have $X_{ij} = X_{ji}$.
X^+_{ij}	`('C',i,j)`	`0 <= i,j < 24, i != j`; we have $X^+_{ij} = X^+_{ji}$.
$X^+_{d\cdot\delta}$	`('T',o,s)`	`o` and `s` can be obtained as follows. Let, `x = SubOctad(d, delta)`. Then `x.vector_tuples()` returns the tuple `(1, 'T', o, s)`. $d \in \mathcal{P}$, $\delta \in \mathcal{C}^*$, d an octad, $\delta \subset d$, δ even. We have `0 <= o < 759`, `0 <= s < 64`.
$X^+_{d\cdot i}$	`('X',d,i)`	`0 <= d < 2048`, `0 <= i < 24`, `d` represents the element `PLoop(d)` of the Parker loop $\mathcal{P}$
$d^+ \otimes_1 i$	`('Z',d,i)`	`d` and `i` as in case `('X',d,i)`
$d^- \otimes_1 i$	`('Y',d,i)`	`d` and `i` as in case `('X',d,i)`

Remarks

- The space ρ_p is an orthogonal space. Two basis vectors are orthogonal except when equal or opposite. The basis vectors $(ij)_1$ have norm `2` in case $i \neq j$. All other basis vectors have norm `1`.
- In the tuple `('T',o,s)` the integers `o` and `s` describe the instance `SubOctad(o, s)` of class *XLeech2*. Here `o` and `s` may be anything that is accepted as input for `SubOctad(o, s)`
- In the tuples `('X',d,i)`, `('Y',d,i)`, `('Z',d,i)`, the input `d` may be an instance of class *PLoop* or anything such that `PLoop(d)` is an instance of class *PLoop*. For an instance `d` of class *PLoop* we have:
  - `('X',d,i) == -('X',-d,i) == ('X',~d,i)` ,
  - `('Y',d,i) == -('Y',-d,i) == -('Y',~d,i)` ,
  - `('Z',d,i) == -('Z',-d,i) == ('Z',~d,i)` .

  The value of `('X',~d,i)` has been set by convention; the other equations are forced.

- The basis vector $d^+ \otimes_1 i$ in [Sey20] is equal to the basis vector $(-1)^{|d/4|} \cdot d^+ \otimes_1 i$ in [Con85]. This modification simplifies the formulas for the operation of the element ξ of $\mathbb{M}$ on ρ_p.
- A first or second index following a tag may be `'r'`, indicating a random integral index. Any omitted index after a given index `'r'` is interpreted as `'r'`. For tags `'A', 'B', 'C', 'T'` the second index should be `'r'` or omitted if the first index is `'r'`, since in these cases the two indices are not sufficiently independent.

1.5.3 Representing a vector as a list of tuples

For creating an arbitrary (but yet sparse) vector in ρ_p the user may call `MMVector(p, data_list)`, where `data_list` is a list of tuples `(factor, tag, i0, i1)`. Here `(tag, i0, i1)` describes a unit vector and the (optional) integer `factor` is a scalar factor for that unit vector. Then the constructor returns the linear combination of the unit vectors given in the list.

In order to generate a random multiple of a basis vector, inside a list of tuples, the (optional) component `factor` of a tuple can be set to one of the following values:

- `'u'`: equivalent to `1`
- `'s'`: a factor `1` or `-1` selected at random
- `'n'`: a random nonzerodivisor modulo `p`
- `'r'`: a random integer modulo `p`

1.5.4 Operations on vector in the representation ρ_p

Vector addition, subtraction and scalar multiplication can be done with operators `+`, `-` and `*` as usual. Elements of the monster group operate on vectors by right multiplication. Only vectors modulo the same characteristic `p` can be added.

The following code example generates an element `g` of the monster group and a vector `v` in the vector space ρ_3 and displays the result `g * v` of the operation of `g` on `v`. Note that a basis vector `(tag, i0, i1)` is displayed in the form `tag_i0_i1`. For a displayed index `i0` or `i1` the suffix `h` means hexadecimal notation.

```
>>> # An instance of class MM represents an element of the monster
>>> from mmgroup import MM
>>> # Function MMV is used to create a representation space
>>> from mmgroup import MMV
>>> # Create representation space V for the monster (modulo 3)
>>> V = MMV(3)
>>> # Create an element g of the monster group
>>> g = MM([('d', 0x123), ('p', 217821225)])
>>> # Create a vector v in the representation space V
>>> v = V([('T', 712, 13), (2, 'X', 0x345, 13)])
>>> # Let g operate on v by right multiplication
>>> print(v * g)
MV<3;-T_444_0fh+X_244h_11>
```

1.5.5 Special tags for creating vectors in the representation ρ_p

Apart from the standard tags `A`, `B`, `C`, `T`, `X`, `Y`, and `Z`, the constructor of class *`MMVector`* accepts a variety of special tags. Details are given in the following list:

Table 18: Special tags for constructing a vector

tag, i0, i1	Evaluates to
`('D', i0)`	Shorthand for `('A', i0, i0)`
`('I', i0, i1)`	Shorthand for the sum of the basis vectors `('A', i0, i0)` + `('A', i1, i1)` - `('A', i0, i1)` - 2 `('B', i0, i1)`, see remark below.
`('J', i0, i1)`	Shorthand for the sum of the basis vectors `('A', i0, i0)` + `('A', i1, i1)` - `('A', i0, i1)` + 2 `('B', i0, i1)`, see remark below.
`('U')`	Tag `'U'` (without any further parameters) stands for the sum `('A', 0, 0)` + `('A', 1, 1)` + ... + `('A', 23, 23)`.
`('E', i)`	This is the basis vector with *linear* index `i`. See the following subsection for details.
`('S', data)`	Here `data` is an array-like object (as defined in the numpy package) that encodes a vector in sparse representation as an array of unsigned 32-bit integers. This is for internal use. For details see the following subsection: Sparse representation of vectors in ρ_p.
`('V', data)`	Here `data` is an array like object (as defined in the numpy package) that encodes a one-dimensional array of integers of length 196884. The order of the entries is as in the next section.
`('R')`	Tag `'R'` (without any further parameters) stands for the generation of a uniform distributed random vector.
`(i, v)`, `i` integer	Here `i` is an integer and `v` is a vector, i.e.an instance of class *`MMVector`*. Then the vector `i * v` is generated. This is useful for extending the modulus `p` of a vector, see remark below.

Remarks

The vectors labelled by `('I', i0, i1)` and `('J', i0, i1)` are axes of the elements x_δ and $x_{-1}x_\delta$ of the monster, respectively, see [Con85]. The centralizers of these elements and also of their axes in the monster have structure $2 \cdot B$, where B is the Baby Monster, see [Asc86], [Con85], [Iva09] for background. E.g. the axes v^+ and v^- in [Sey22] may be obtained as `MMVector(15, 'I', 2, 3)` and `MMVector(15, 'J', 2, 3)`, respectively.

The modulus of a vector can be extended as follows. Assume that `v` is a vector in ρ_3, given as an instance of class *`MMVector`*. Then `w = 5 * v` is a well-defined vector in ρ_{15}. Vector `w` can be constructed as follows: `V15 = MMV(15); w = V15(5, v)`. Note that the construction `w = V15(5*v)` leads to an error, since `5*v` is defined modulo `3`, but not modulo `15`.

1.5.6 Linear order of the basis vectors

By construction of the representation ρ_p, it is most natural to use a tuple `(tag, i0, i1)` for indexing a basis vector of ρ_p. There is also a linear order of these entries. Such a linear order is required e.g. for expressing a vector in ρ_p as an array of integers modulo p. Therefore an index given by a tuple `(tag, i0, i1)` is mapped to linear index `0 <= i < 196884`. Legal standard tuples are listed in the following table:

Table 19: Conditions for indices `i0`, `i1`

Tag	Condition for `i0`	Condition for `i1`
`'A'`	`0 <= i0 < 24`	`0 <= i1 < 24`
`'B', 'C'`	`0 <= i0 < 24`	`0 <= i1 < 24`, `i1 != i0`
`'T'`	`0 <= i0 < 759`	`0 <= i1 < 64`
`'X', 'Y', 'Z'`	`0 <= i0 < 2048`	`0 <= i1 < 24`

For `tag = 'A', 'B', 'C'`, we also have `(tag, i0, i1) == (tag, i1, i0)`.

The linear order of the tuples `(tag, i0, i1)` is as follows:

offset `0`:

```
(A,0,0), (A,1,1), ..., (A,23,23)
```

offset `24`:

```
(A,1,0),
(A,2,0), (A,2,1)
...
(A,23,0), (A,23,1), ..., (A,23,22)
```

offset `300`:

Tuples `(B,i0,i1)`, same order as tuples `(A,i0,i1)` for `i0 > i1`

offset `576`:

Tuples `(C,i0,i1)`, same order as tuples `(A,i0,i1)` for `i0 > i1`

offset `852`:

```
(T, 0,0), (T, 0,1), ..., (T, 0,63)
(T, 1,0), (T, 1,1), ..., (T, 1,63)
...
(T,758,0), (T,758,1), ..., (T,758,63)
```

offset `49428`:

```
(X, 0,0), (X, 0,1), ..., (X, 0,23)
(X, 1,0), (X, 1,1), ..., (X, 1,23)
...
(X,2047,0), (X,2047,1), ..., (X,2047,23)
```

offset `98580`:

Tuples `(Z,i0,i1)`, same order as corresponding tuples `(X,i0,i1)`

offset `147732`:

Tuples `(Y,i0,i1)`, same order as corresponding tuples `(X,i0,i1)`

1.5.7 Sparse representation of vectors in ρ_p

Internally, a vector ρ_p is sometimes given in *sparse* representation. This is useful for sparse vectors containing many zero entries. In *sparse* representation the vector is given as an array of unsigned 32-bit integers. Here each entry of the array encodes a multiple of the basis vector. Such an entry it interpreted as follows:

Table 20: Bit fields in an entry of a sparse representation

Component	`tag`	`i0`	`i1`	`factor`
Bits	`27 - 25`	`24 - 14`	`13 - 8`	`7 - 0`

This corresponds to the tuple `(factor, tag, i0, i1)` which denotes a multiple of a basis vector as described in subsection *Representing a vector as a list of tuples*.

Tags are mapped to integers as follows:

Table 21: Mapping of integers to tags

`'A'`	`'B'`	`'C'`	`'T'`	`'X'`	`'Z'`	`'Y'`
`1`	`2`	`3`	`4`	`5`	`6`	`7`

1.5.8 Other data types accepted as tags for vectors in ρ_p

Some data types are accepted as tags and interpreted as described in the following table. In this case parameters *i0*, *i1* after a tag must not be set.

Table 22: Legal types for constructing a vector

type	Evaluates to
List of tuples	This evaluates to a vector as described in subsection *Representing a vector as a list of tuples*. An entry of such a list may also be an instance of class `MMVector` or a string.
class `MMVector`	A deep copy of the given vector is returned.
class `XLeech2`	If `tag` is of type `XLeech2` then the (possibly negative) basis vector corresponding to that `tag` in Q_{x0} is created.
`str`	For an vector `v` in `V` we have `V(str(v)) == v`. This is helpful for rereading printed vectors.
`('Axis', g)`	If a pair containing the string `'Axis'` and a 2A involution `g` in the Monster group is given then we contruct the axis corresponding to that involution (with norm 8) as described in [Con85]. Here `g` should be an instance of class `XLeech2` or `MM`. A integer or string `g` describes the element `MM('q', g)`.

1.5.9 Python classes implementing the representation of the Monster group

class `mmgroup.`**`MMVector`**(*p*, *tag=0*, *i0=None*, *i1=None*)

Models a vector in a representation of the monster group.

The construction of vectors in the representations ρ_p of the monster group (using e.g. the constructor of this class) is documented at the beginning of section *The representation of the Monster group*. There a basis vector of that representation is described as a tuple `(tag, i0, i1)`, where `tag` is a single capital letter and `i0`, `i1` are integers. So we may invoke the constructor of this class as follows:

Parameters

- `p` – Modulus `p` of the representation ρ_p.
- `tag` – Tag of the unit vector to be constructed. In the standard case this is a single capital letter describing the type of the basis vector.
- `i0` – First index in the tuple describing the basis vector.
- `i2` – Second index in the tuple describing the basis vector.

Linear operations can be used for for creating linear combination of unit vectors. This is just the simplest case of the construction of a vector in ρ_p. Section *The representation of the Monster group* contains a large number of further possibilities for creating such vectors.

Addition and subtraction of vectors in the same space work as usual. This is also the case for scalar multiplication with integers. An element `g` of the monster group (given as an instance of class *MM*) operates on a vector `v` (given as an instance of this class) by right multiplication.

An entry of a vector `v` (given as an instance of this class) may be addressed as `v[tag, i0, i1]`, where `tag` is a single letter in the string `ABCTXYZD` and `i0` and `i1` are integers, such that the tuple `(tag, i0, i1)` describes a basis vector. Getting and setting entries of a vector is as in the `numpy` package. Here `i0` and `i1` may also be slices of integers in the same way as in `numpy` arrays. Depending on the `tag`, the value `i0` or `i1` may also be a single instance of the appropriate class, as described in the remarks after table *Basis vectors of the representation of the monster* .

The entries of vector `v` also have a linear order as described at the beginning of section *The representation of the Monster group*. Here `v['E', i]` is the `i`-th entry in that order. Index `i` may also be a slice of integers in the same way as in a one-dimensional `numpy` array.

The internal representation of a vector `v` in this class is not part of the public interface. Use `v['E']` to convert `v` to an one-dimensional array of `8`-bit integers of length 196884.

`as_sparse()`

Return vector in sparse representation

The sparse representation of a vector is returned as a one-dimensional numpy array of 32-bit integers. Here each entry of that array represents a nonzero component of the vector. Such an entry it interpreted as described in method `from_sparse` of class *MMSpace*.

Entries with value zero are dropped.

`as_tuples()`

Return vector in tuple representation

The function returns a list of tuples `(factor, tag, i0, i1)`. Here `(tag, i0, i1)` describes a basis vector and `value` is the coordinate of that vector.

Entries with coordinate zero are dropped.

`mul_exp`(*g*, *e=1*, *break_g=False*)

Multiply the vector with `g ** e` inplace

Here `g` is an element of the monster group represented as an instance of class *MM* and `e` is an integer. The vector is updated and the updated vector is returned.

Afterwards, the vector contains an attribute `last_timing` containing the run time of this operation in seconds. This is useful for benchmarking.

By default, we try to simplify the expression `g ** e` before multiplying it with the vector. If `break_g` is set, we always do `abs(e)` multiplications with `g` or with its inverse.

projection(**args*)

Return projection of the vector `v` onto a subspace.

Each argument describes a subspace of this space `V`. The projection of the vector to the space `W` spanned by all these subspaces is returned.

In the current version each argument must be a tuple `(tag, i0, i1)` that describes a subspace generated by a basis vector. A legal `tag` is letter in the string `ABCTXYZD`, as explained in table *Basis vectors of the representation of the monster* and class *`MMSpace`*.

class `mmgroup.`**MMSpace**(**args*, ***kwds*)

Models a `196884`-dimensional representation of the monster group

This class contains a collection of functions for manipulating vectors in the representation ρ_p of the monster group. Such vectors are instances of class *`MMVector`*. Most of these function are used implicitly in the operators applied to these vectors.

Some of these function may be helpful for the user; so we document them in the *mmgroup API reference*.

static index_to_short(*tag*, *i0=-1*, *i1=-1*)

Convert index to a short Leech lattice vector

If `tag` is an integer, this is interpreted a linear index for a basis vector in the representation ρ_p as in method `tuple_to_index`. Otherwise the tuple `(tag, i0, i1)` is interpreted a standard index for such a basis vector. If `tag` is an instance of class *`MMVector`*, which is a nonzero multiple of a basis vector, then that basis vector is taken.

Some but not all of these basis vectors correspond to short vector in the Leech lattice up to sign. If this is the case then the function returns a short vector of the Leech lattice as a numpy array of signed `32`-bit integers of length `24`. The norm of the returned vector is `32`.

The function raises ValueError if the basis vector in ρ_p does not correspond to a short vector.

The sign of the short vector is undefined, but two calls referring to the same basis vector return the same short vector.

classmethod index_to_tuple(*index*)

Convert linear index to tuple `(tag, i0, i1)`

This method reverses the effect of method `tuple_to_index`. Given a linear index `0 <= i < 196884` for a basis vector, the function returns that index as a tuple `(tag, i0, i1)` with `tags` one letter of the string `"ABCTXYZ"` and integers `i0`, `i1`.

See method `tuple_to_index` for details.

If `index` is an instance of class *`MMVector`*, which is a nonzero multiple of a basis vector, then the index of that basis vector is taken.

classmethod tuple_to_index(*tag*, *i0=-1*, *i1=-1*)

Convert tuple `(tag, i0, i1)` to a linear index

Remarks:

The tuple `('D', i0)` is accepted as a shorthand for `('A', i0,i0)`.

A tuple `('E', i0)` means a linear index `i0`, i.e. this method returns `i0` on input `('E', i0)`, for `0 <= i0 < 196884`.

If `tag` is an instance of class *`MMVector`*, which is a nonzero multiple of a basis vector, then the linear index corresponding to that basis vector is returned.

1.5.10 Auxiliary functions for the representation of the Monster group

`mmgroup.characteristics()`

Return list of all *characteristics* `p` supported

`mmgroup.MMV(p)`

Return an object corresponding to the space ρ_p

Here characteristic `p` is as in the constructor of class *MM*. Thus the sequence

```
>>> V = MMV(p)
>>> v = V(tag, i0, i1)
```

returns the same vector in ρ_p as the statement `MMVector(p, tag, i0, i1)`.

Function `characteristics` in module `mmgroup` returns the list of legal values for the characteristic `p`.

Technically, `MMV(p)` is the partial application of the constructor of class *MMVector* to the number `p`.

`mmgroup.mmv_scalprod(v1, v2)`

Return scalar product of two vectors

The function returns the scalar product of the vectors v_1 and v_2. These two vectors must be instances of class *MMVector*; and they must be vectors in the same representation ρ_p.

`mmgroup.order_vector()`

Return the precomputed order vector

The function returns the precomputed order vector v_1 (which is an element of the representation ρ_{15} of the monster) as an instance of class *MMVector*.

The properties required for such a vector v_1 are defined in [Sey22].

1.5.11 Long-term stable storage of vectors of the representation

The representations ρ_p of the Monster are considered as auxiliary tools for computing in the Monster. We do not guarantee the interoperability of vectors in these representations between different versions of the *mmgroup* package. The internal representation of such vectors is likely to change in the near future. A user with a serious need for such interoperability can consult the author for advice.

1.6 The subgroup G_{x0} of the Monster and the Clifford group

The section describes the fast computation in a certain subgroup G_{x0} of structure $2^{1+24}.Co_1$ of the Monster $\mathbb{M}$ in detail. A person who simply wants do do calculations in the Monster group need not read this section.

1.6.1 Introduction

In Conway's construction [Con85] the Monster $\mathbb{M}$ has a subgroup G_{x0} of structure $2_+^{1+24}.\mathrm{Co}_1$. There G_{x0} is constructed as a diagonal product of the two groups Co_0 of structure $2.\mathrm{Co}_1$ and $G(4096_x)$. $G(4096_x)$ is also of structure $2_+^{1+24}.\mathrm{Co}_1$ but not isomorphic to G_{x0}. Here 2_+^{1+24} means an extraspecial group of plus type. Co_0, and Co_1 are the automorphism groups of the 24-dimensional Leech lattice and of the Leech lattice modulo 2, see e.g. [CCN+85] or [CS99] for background.

Computation in Co_0 is easy since that group has a 24-dimensional rational representation. The smallest real representation of the group $G(4096_x)$ has dimension 4096, so naive computation in that representation is rather inefficient.

The group $G(4096_x)$ is a subgroup of the real Clifford group $\mathcal{C}_{12}$. The real Clifford group $\mathcal{C}_n$ of structure $2_+^{1+2n}.\mathrm{GO}_{2n}^+(2)$ is defined e.g. in [NRS01]. $\mathcal{C}_{12}$ is a subgroup of the complex Clifford group $\mathcal{X}_n$ of structure $\frac{1}{2}(2_+^{1+2n} \times Z_8).\mathrm{Sp}_{2n}(2)$, which is also defined in [NRS01]. Here a factor $\frac{1}{2}$ means identification of central subgroups of order 2 of the factors of a direct product. $\mathrm{GO}_{2n}^+(2)$ and $\mathrm{Sp}_{2n}(2)$ are the orthogonal subgroup (of $+$-type) and the symplectic subgroup of the group $GL_{2n}(2)$ of invertible $2n \times 2n$-matrices with entries in $\mathbb{F}_2$, as defined in [CCN+85].

Effective computation in the group $\mathcal{X}_n$ has received a lot of attention from the theory of quantum computation, see e.g. [AG04]. In the next section we present efficient algorithms for computing in $\mathcal{X}_n$ and in its 2^n dimensional complex representation.

1.6.2 Computation in the Clifford group

In this section we present effective algorithms for computing in the complex Clifford group of structure $\frac{1}{2}(2_+^{1+2n} \times Z_8).\mathrm{Sp}_{2n}(2)$ defined in [NRS01]. Here a factor $\frac{1}{2}$ means identification of central subgroups of order 2 of the factors of a direct product as in [CCN+85].

The complex Clifford group $\mathcal{X}_n$ has a unitary complex representation of dimension 2^n which has been studied in the theory of quantum computation, see e.g. [AG04]. For calculations in the monster group it would be sufficient to deal with the subgroup $\mathcal{C}_n$ of $\mathcal{X}_n$ consisting of the real matrices of that representation. However, the extra effort for extending the real to the complex case is marginal, and using our algorithms for the complex Clifford group might be useful in the theory of quantum computation.

Usually, in quantum physics it suffices to calculate in a unitary group up to a scalar multiple of the unit matrix. Therefore we cannot simply cut an paste existing algorithms for calculating in $\mathcal{X}_n$ used in quantum physics. We keep the following exposition independent of the theory of quantum computation, but we do not hide the fact that the main ideas are strongly influenced by that theory.

We present an algorithm that performs matrix multiplication in $\mathcal{X}_n$ in $O(n)^3$ bit operations. Therefore we store an element of $\mathcal{X}_n$ in $O(n)^2$ bits. In case $n = 12$ this is considerably more efficient than dealing with complex 4096×4096 matrices.

We will introduce certain complex-valued functions defined on a Boolean vector space $\mathbb{F}_2^n$ which we will call *quadratic mappings*. Such a function has a natural interpretation as a vector in $\mathbb{C}^{2^n}$. It will turn out that quadratic mappings are closed under tensor products and under tensor contraction. Since matrix multiplication is just a special case of tensor contraction, we may consider quadratic mappings on $\mathbb{F}_2^n \times \mathbb{F}_2^n$ as a monoid of $2^n \times 2^n$ matrices closed under matrix multiplication. It turns out that the unitary matrices in this monoid are just a representation of the Clifford group $\mathcal{X}_n$.

Quadratic mappings

Let $V = \mathbb{F}_2^n$ be a Boolean vector space and $\mathbb{T}$ be the unit circle in the set $\mathbb{C}$ of the complex numbers. Then $\mathbb{F}_2^n$ is an additive and $\mathbb{T}$ is a multiplicative Abelian group. For any mapping $f : V \rightarrow \mathbb{T}$ define the mapping $\Delta f : V \times V \rightarrow \mathbb{T}$ by

$$\Delta f(x, y) = f(x + y) \cdot f(x)^{-1} \cdot f(y)^{-1} \cdot f(0).$$

A mapping $g : V \rightarrow \mathbb{T}$ is bilinear if

$$g(x_1 + x_2, x_3) = g(x_1, x_3) \cdot g(x_2, x_3),$$
$$g(x_1, x_2 + x_3) = g(x_1, x_3) \cdot g(x_1, x_3).$$

Then we obviously have $g(x_1, x_2) = \pm 1$ and $g(0, x_2) = g(x_1, 0) = 1$. Thus there is a unique symmetric bilinear form $\beta(g) : V \times V \rightarrow \mathbb{F}_2$ with $g(x_1, x_2) = (-1)^{\beta(g)(x_1,x_2)}$. A function $q : V \rightarrow \mathbb{T}$ is called *quadratic* if $q(0) = 1$ and Δq is bilinear. For a quadratic function q all functional values of q are fourth roots of unity. We write $\beta(q)$ for $\beta(\Delta q)$. Put

$$R_8 = \{2^{e/2} \cdot w \mid e \in \mathbb{Z}, w \in \mathbb{C}, w^8 = 1\} \cup \{0\} \ .$$

Let $e \in R_8$, let $q : \mathbb{F}_2^m \rightarrow \mathbb{T}$ be a quadratic function, and let $a : \mathbb{F}_2^m \rightarrow \mathbb{F}_2^n$ be an affine mapping. Then we define a function $f = f(e, a, q) : \mathbb{F}_2^n \rightarrow \mathbb{C}$ by

$$f(e, a, q)(x) = e \cdot \sum_{\{y \in \mathbb{F}_2^m \mid a(y) = x\}} q(y). \tag{1.1}$$

We call a function $f(e, a, q)$ satisfying (1.1) a *quadratic mapping*. We call the triple (e, a, q) a *representation* of the quadratic mapping $f(e, a, q)$. Occasionally we also consider quadratic mappings $f(e, a, q)$ where q is a scalar multiple of a quadratic function with $q(0) \in R_8$. We sometimes abbreviate $f(e, a, q)(x)$ to $f(x)$ if the meaning of e, a, q is clear from the context.

The following lemma is a direct consequence of the definition of a quadratic mapping.

Lemma 1:

Let $g, g_1 : V = \mathbb{F}_2^n \rightarrow \mathbb{C}$ be quadratic mappings. Then

- For any affine mapping $a : \mathbb{F}_2^{n'} \rightarrow \mathbb{F}_2^n$ the composition $g \circ a$ given by $x \mapsto g(a(x))$ is a quadratic mapping on $\mathbb{F}_2^{n'}$.

  Hence the definition of a quadratic mapping on the affine space V is invariant to translations and to basis transformations of V.

- The product $g \cdot g_1$ of the functions g and g_1 is a quadratic mapping on V.
- For any any affine subspace W of V restriction of g to W, and also characteristic function $\chi_W : V \rightarrow V$ of the subspace W is a quadratic mapping.
- For any linear subspace W of V the function $g^{(W)} : V/W \rightarrow \mathbb{C}$ given by $x \mapsto \sum_{y \in W} g(x + y)$ is a quadratic mapping.

A function $f : \mathbb{F}_2 \rightarrow \mathbb{C}$ has a natural interpretation as a vector in $\mathbb{C}^2$. Similarly, a function $g : \mathbb{F}_2^n \rightarrow \mathbb{C}$ has a natural interpretation as a tensor in the tensor product $(\mathbb{C}^2)^{\otimes n}$. If $\mathbb{C}^2$ has basis (b_0, b_1) then the tensor corresponding to g has coordinate $g(i_1, \ldots, i_n)$ at the basis vector $b_{i_1} \otimes \ldots \otimes b_{i_n}$ of $(\mathbb{C}^2)^{\otimes n}$. We call g the *coordinate function* of the tensor. If g is a function $\mathbb{F}_2^n \rightarrow \mathbb{C}$ and $x_j \in \mathbb{F}_2^{n_j}$ holds with $\sum_{j=1}^k n_j = n$ then we put $g(x_1, \ldots, x_k)$ = $g(x)$, where x is the concatenation of the bit vectors $x_1, \ldots, x_k$.

A vector (or a tensor) in a space over $\mathbb{F}_2^n$ is called a *quadratic state vector* (or a quadratic state tensor) if its coordinate function is a quadratic mapping. Using Lemma 1 we can easily show that tensor products and tensor contractions of quadratic state vectors are quadratic state vectors. As we shall see later, they can easily be computed in practice. A matrix is a special kind of a tensor, and matrix multiplication is a special case of the contraction of a tensor product. Thus the matrix product of two quadratic state matrices is also a quadratic state matrix.

For any affine mapping $a : \mathbb{F}_2^m \rightarrow \mathbb{F}_2^n$ there is a unique linear mapping $l(a)$ that differs from a by the constant $a(0)$. We write $\ker a$ for $\ker l(a)$. We call a representation (e, a, q) of a quadratic mapping *injective* if $\ker a = 0$. We have:

Lemma 2:

Every quadratic mapping has an injective representation.

Proof

Let $e \in R_8, a : \mathbb{F}_2^m \rightarrow \mathbb{F}_2^n$, and q a quadratic function on $\mathbb{F}_2^m$ such that $g = f(e, a, q)$ and m is minimal. Assume $\ker a \neq 0$. Then we construct a tuple (e', a', q') with $g = f(e', a', q')$, such that the domain of a' and of q' is a proper affine subspace of the domain of a.

Let $v \in \ker A \setminus \{0\}$ and W be a subspace of $\mathbb{F}_2^m$ with $\langle v \rangle \oplus W = \mathbb{F}_2^m$. Then for $w \in W$ we have $q(w + v) = q(w) \cdot (-1)^{h(w)} \cdot r$, with h a linear form on W given by $h(w) = \beta(q)(v, w)$ and $r = q(v)/q(0)$ a fourth root of unity. Since $v \in \ker A$, we have

$$f(e, a, q)(x) = e \cdot \sum_{\{y \in W | a(y) = x\}} q(y) + q(y + v) = e \cdot \sum_{\{y \in W | a(y) = x\}} q(y) t(h(y)) \ ,$$

where $t(z) = 1 + r \cdot (-1)^z$ for $z \in \mathbb{F}_2$. Let W' be the support of $t \circ h$, i.e. $W' = \{y \in W \mid t(h(y)) \neq 0\}$. In case $W' = \emptyset$ we have $f(e, a, q) = 0$. Otherwise it suffices to show that W' is an affine subspace of W, and that the restriction of $t \circ h$ to W' is a quadratic function up to a scalar multiple in R_8.

Since r is a fourth root of unity, $t(0)$ and $t(1)$ are in R_8. If h is zero then $t \circ h$ is a constant function on W.

If r is an imaginary fourth root of unity then $t(0) \neq 0$ and $t(1)/t(0) = \pm\sqrt{-1}$. Thus t is a quadratic function on $\mathbb{F}_2$ up to the factor $t(0)$. Since h is linear, $t \circ h$ is also a quadratic function on W.

If $r = \pm 1$, $h \neq 0$, then the function $t \circ h$ is constant on W', and we have $W' = \ker h$ if $r = 1$ and $W' = W \setminus \ker h$ if $r = -1$.

q.e.d.

One of the basic challenges in this section is the effective computation of an injective representation of a quadratic mapping from an arbitrary representation of that mapping. The proof of Lemma 2 suggests that this job can be done by using standard methods of linear algebra, mainly over $\mathbb{F}_2$.

The complex Clifford group $\mathcal{X}_n$ is a group which is defined in [AG04] and [NRS01]. It has a unitary representation in $(\mathbb{C}^2)^{\otimes n}$.

Lemma 3

The unitary complex quadratic state matrices representing endomorphisms of the space $\mathbb{C}^{2^n}$ make up a representation of the complex Clifford group $\mathcal{X}_n$.

Sketch Proof

It is easy to see that all generators of $\mathcal{X}_n$ in [NRS01] are unitary complex quadratic state matrices. The group $\mathcal{X}'_n$ of such matrices is closed under multiplication. It is obviously closed under computing the inverse, which is the conjugate transpose for a unitary matrix. Thus $\mathcal{X}_n$ is a subgroup of $\mathcal{X}'_n$. By Lemma 3 the group $\mathcal{X}'_n$ is finite. By Theorem 6.5 in [NRS01] all finite supergroups of $\mathcal{X}_n$ in the unitary group are generated by $\mathcal{X}_n$ and a scalar multiple of the unit matrix by a root of unity. Comparing the scalar multiples of the unit matrix in $\mathcal{X}_n$ and $\mathcal{X}'_n$ yields $\mathcal{X}_n$ = $\mathcal{X}'_n$.

q.e.d.

Background from the theory of quantum computing

In the theory of quantum computation a state vector representing the state of n qubits can be written as a vector in $(\mathbb{C}^2)^{\otimes n}$, where the 2^n basis vectors of that space are labelled by the elements of $\mathbb{F}_2^n$. Here the n qubits correspond to the n factors $\mathbb{F}_2$ of $\mathbb{F}_2^n$. In [AG04] the unit vectors which are also quadratic state vectors are called *stabilizer states*. By Lemma 2 the product of a stabilizer state with a unitary quadratic state matrix is a stabilizer state. Thus Lemma 3 implies that the Clifford group $\mathcal{X}_n$ stabilizes the stabilizer state vectors. In the theory of quantum computation this fact is known as the Gottesman-Knill theorem.

In [AG04] there is a fast algorithm for calculating in the Clifford group $\mathcal{X}_n$. As usual in quantum theory, this algorithm ignores scalar factors in the matrix representation of $\mathcal{X}_n$. This means that we have to create our own algorithm for computing in $\mathcal{X}_n$.

Implementation of quadratic mappings

Let $g = f(e, a, q) : \mathbb{F}_2^n \rightarrow \mathbb{C}$ with $e \in R_8$, $a : \mathbb{F}_2^m \rightarrow \mathbb{F}_2^n$ an affine mapping, and $q : \mathbb{F}_2^m \rightarrow \mathbb{T}$. We implement the quadratic mapping g as a triple (e, A, Q).

Here A is an $(m+1) \times n$ bit matrix representing a, and Q is a symmetric $(m+1) \times (m+1)$ bit matrix representing q. All vector and matrix indices start with 0 as usual in the C language. For $x = (x_1, \ldots, x_m) \in \mathbb{F}_2^m$, $x_0 = 1$ and bit matrices A, Q as above we put:

$$a(x) = (x_0, \ldots, x_m) \cdot A \quad , \quad q(x) = \exp\left(\pi\sqrt{-1}/2 \cdot \sum_{j,k=0}^{m} Q_{j,k} x_j x_k\right) .$$

We write $\sqrt{-1}$ for the imaginary unit, and we use the letters $i, j, \ldots$ as indices.

The user may consider a quadratic mapping $g : \mathbb{F}_2^n \rightarrow \mathbb{C}$ as a 2^n-dimensional complex vector, regardless of its representation. In C or python its is convenient to write `g[i]` for the `i`-th component of such vector `g`, and to assume that the index $i = \sum_{j=0}^{n-1} 2^j \cdot i_j$, $i_j \in \{0, 1\}$ denotes the bit vector $(i_{n-1}, \ldots i_0)$ in $\mathbb{F}_2^n$. In accordance with that convention we label the entries of the $m + 1 \times n$ bit matrix A as follows:

$$A = \begin{pmatrix} A_{n-1,0} & \ldots & A_{0,0} \\ \vdots & & \vdots \\ A_{n-1,m} & \ldots & A_{0,m} \end{pmatrix} .$$

Any affine mapping $a : \mathbb{F}_2^m \rightarrow \mathbb{F}_2^n$ can be written as $(x_1, \ldots, x_m) \mapsto (1, x_1, \ldots, x_m) \cdot A$ for a suitable $(m+1) \times n$ bit matrix A as above.

Wie label the entries of the symmetric $m + 1 \times m + 1$ bit matrix Q as usual. We stress that all bits $Q_{j,k}, x_j, x_k$ in sum in the expression for $q(x)$ must be interpreted as integers equal to 0 or 1 (modulo 4). But since Q is a symmetric matrix, it suffices to know the off-diagonal entries of Q modulo 2 if we assume $Q_{j,k} = Q_{k,j}$. Due to the condition $q(0) = 1$ we only consider matrices Q with $Q_{0,0} = 0$. It is easy to check that we can encode any quadratic function $q : \mathbb{F}_2^m \rightarrow \mathbb{T}$ as a symmetric $(m+1) \times (m+1)$ bit matrix Q (with $Q_{0,0} = 0$) uniquely as above.

Given e and matrices A, Q as above, we define $f(e, A, Q) = f(e, a, q)$, where a and q are as in the last equation.

We encode a number $e \in R_8 \setminus \{0\}$ as an integer e' such that $e = \exp(e'\pi\sqrt{-1}/4) \cdot 2^{\lfloor e'/16 \rfloor/2}$, with $\lfloor x \rfloor$ the greatest integer $\leq x$. We encode the constant quadratic mapping 0 as a matrix A with zero rows.

Changing the representation of a quadratic mapping

The representation $f(e, A, Q)$ of a quadratic mapping q is not unique. In this section we define some elementary operations $T_{i,j}, X_{i,j}$ on a triple (e, a, Q) that to not change the quadratic mapping $f(e, A, Q)$. In the next section we will use these operations to reduce the representation $f(e, A, Q)$ to a standard form.

Let $(b_1, \ldots, b_m)$ be the standard basis of $\mathbb{F}_2^m$. For $1 \leq i, j \leq m, i \neq j$, define the linear transformation $T_{i,j} : \mathbb{F}_2^m \rightarrow \mathbb{F}_2^m$ by:

$$T_{i,j}(b_j) = b_j + b_i \, , \, T_{i,j}(b_k) = b_k \quad \text{for} \quad k \neq j \; .$$

We also define the (affine) translation $T_{0,j} : \mathbb{F}_2^m \rightarrow \mathbb{F}_2^m$ by $T_{0,j}(x) = x + b_j$ for all $x \in \mathbb{F}_2^m$.

Let q' be the quadratic function $q \circ T_{i,j}$ given by $x \mapsto q(T_{i,j}(x))$. Let Q' be the symmetric bit matrix representing q'. Then for $i, j > 0, i \neq j$ we have

$$\begin{array}{rcll} Q'_{i,0} = Q'_{0,i} & = & Q_{i,0} + Q_{j,0} + Q_{i,j} + Q_{i,i} \cdot Q_{j,j} \, , & \\ Q'_{i,i} & = & Q'_{i,i} + Q_{j,j} \, , & \\ Q'_{i,k} = Q'_{k,i} & = & Q_{i,k} + Q_{j,k} & \text{for} \quad k > 0, k \neq i \, , \\ Q'_{k,l} & = & Q_{k,l} & \text{for all other pairs} \quad (k,l) \; . \end{array}$$

The quadratic function $q \circ T_{0,j}, \; j > 0$ is equal to $e \cdot q'$ with $e = \exp(\pi \sqrt{-1} \cdot (Q_{0,i} + Q_{i,i}/2))$. Here q' is the quadratic function represented by the bit matrix Q' with

$$\begin{array}{rcll} Q'_{0,0} & = & 0 \, , & \\ Q'_{0,k} = Q'_{k,0} & = & Q_{0,k} + Q_{j,k} & \text{for} \quad k > 0 \, , \\ Q'_{k,l} & = & Q_{k,l} & \text{for all other pairs} \quad (k,l) \; . \end{array}$$

For an affine mapping $a : \mathbb{F}_2^m \rightarrow \mathbb{F}_2^n$ the mapping $a' = a \circ T_{i,j}$ with $i \geq 0, j > 0, j \neq i$ is represented by the matrix A' given by:

$$\begin{array}{rcll} A'(i,l) & = & A(i,l) + A(j,l) \, , & \\ A'(k,l) & = & A(k,l) & \text{for} \quad k \neq j \; . \end{array}$$

This means that adding row and column $j > 0$ to row and and column $i \geq 0$ of a matrix Q representing a quadratic function changes that matrix to a matrix Q' representing $Q \circ T_{i,j}$, up to a scalar factor (which is a fourth root of unity) and some corrections required for the entries $Q'_{0,j}, Q'_{j,0}$ and $Q'_{0,0}$ of Q'. Similarly, adding row j to row i of a matrix A representing an affine mapping from $\mathbb{F}_2^m$ to $\mathbb{F}_2^n$ changes matrix A to a matrix representing the affine mapping $A \circ T_{i,j}$.

We obviously have $f(e, A, Q) = f(e, A \circ T_{i,j}, Q \circ T_{i,j})$. So we may perform a row operation on matrix A, and also a row and a column operation on matrix *Q* without changing *f(e, A, Q)*, (up to a few corrections in line and column 0 of *Q*).

If $f(e, A, Q)$ is in *echelon form*, (as defined in the following section), then it remains in echelon form after applying an operation $T_{i,j}, i < j$.

For $i, j > 0$ let $X_{i,j}$ be the operation on the triple (e, A, Q) that exchanges rows i with row j of A and Q, and also column i with column j of Q. The operation $X_{i,j}$ does not change $f(e, A, Q)$.

Reducing the representation of a quadratic mapping

We call a representation (e, A, Q) of a quadratic mapping *reduced* if the following conditions hold:

- A has no nonzero rows.
- Let A_1 be the matrix obtained from A by prepending the column vector $(1, 0, \ldots, 0)^\top$. Then the leading coefficient of a row of A_1 is always strictly to the right of the leading coefficient of the row above it.
- Each column containing a leading coefficient of a row has zeros in all its other entries.

Here the *leading coefficient* of a row of matrix A is the first nonzero entry in that row. We call the representation (e, A, Q) *echelonized* if only the first two of the three conditions above are satisfied.

Obviously, a reduced or echelonized representation (e, A, Q) also injective. It is easy to see that any quadratic mapping has a unique reduced representation.

In this section we present an algorithm for converting a representation of a quadratic function to a reduced representation. Function `qstate12_reduce` in module `qstate12.c` implements this algorithm.

A matrix is in *row echelon form* if

- All rows consisting of only zeros are at the bottom.
- The leading coefficient of a nonzero row is always strictly to the right of the leading coefficient of the row above it.

A matrix A is in *reduced row echelon* form if it is in echelon form, and each column containing a leading coefficient of a row has zeros in all its other entries. See e.g. https://en.wikipedia.org/wiki/Row_echelon_form .

Let $T_{i,j}, X_{i,j}$ be as in the last section. In order to reduce a representation (e, A, Q) of a quadratic mapping $f(e, A, Q)$ we apply several operations $T_{i,j}, X_{i,j}$ on the components e, A, Q. Neither of these two operations changes $f(e, A, Q)$.

Given A, let $A_1 = A_1(A)$ be obtained from A as above. By applying a sequence of operations $T_{i,j}$, $X_{i,j}$ we may convert A_1 to reduced echelon form. If A_1 is in reduced echelon form and contains no zero rows at the bottom then the representation (e, A, Q) is reduced. Otherwise we use the following algorithm repeatedly to remove the zero rows from the bottom of A.

Let i be the index of the last row of A and assume $A_{i,j} = 0$ for all j.

Let i' be the highest index with $Q_{i,i'} = 1$. If such an i' exists then we add row i' of A to all rows k where $Q_{k,i} = 1$ holds and we adjust Q. Afterwards we have $Q_{i',i} = 1$ for at most one index i'.

Case 1: $Q_{i',i} = 0$ for all i'

The we may delete the last row of A , adjust Q, and double e without changing $f(e, A, Q)$.

Case 2: $Q_{0,i} = 1$

Then $f(e, A, Q)$ is the constant function 0.

Case 3. $Q_{i',i} = 1, 0 < i' < i$

Then we add row i of A to all rows $k \notin \{i, i'\}$ where $Q_{k,i'} = 1$ holds and we adjust Q. Afterwards we have $Q_{k,i'} = Q_{i',k} = Q_{k,i} = Q_{i,k} = 0$ for all $k \notin \{i, i'\}$, $Q_{i,i} = 0$, and $Q_{i,i'} = Q_{i',i} = 1$. So we may delete rows i and i' of A, adjust Q, and double e without changing $f(e, A, Q)$.

Case 4: $Q_{i,i} = 1$

Then $f(e, A, Q)$ is not changed if we delete the last row i of A, adjust Q, and multiply e by $1 + \sqrt{-1}$.

Remark

The algorithm sketched in this subsection yields an alternative proof of Lemma 2.

Extending a quadratic mapping

Let $g : \mathbb{F}_2^n \rightarrow \mathbb{C}$ be a quadratic mapping with $g = f(e, A, Q)$, where A is an $(m+1) \times n$ and Q is an $(m+1) \times (m+1)$ bit matrix. Define $g^{(j)} : \mathbb{F}_2^{n+1} \rightarrow \mathbb{C}$ by

$$g^{(j)}(x_n, \ldots, x_j, x_{j-1}, \ldots, x_0) = g(x_n, \ldots, x_{j+1}, x_{j-1}, \ldots, x_0) \; .$$

So $g^{(j)}$ does not depend on bit j of x.

Then we have $g^{(j)} = f(e, A', Q')$ for matrices A', Q' defined as follows. A' is obtained from A' by first appending a zero row at the bottom, and then inserting a zero column to the right of bit position j. Finally, we change the entry at the bottom of the inserted column from 0 to 1. Q' is obtained from Q' by appending a zero row at the bottom and a zero column at the right. Then we have

$$f(e, A', Q')(x_n, \ldots, x_j, x_{j-1}, \ldots, x_0) = f(e, A, Q)(x_n, \ldots, x_{j+1}, x_{j-1}, \ldots, x_0) \; ,$$

as required. Function `qstate12_extend` in module `qstate12.c` implements the extension of a quadratic mapping.

Products and tensor products of quadratic mappings

Let $g^{(\lambda)}, \lambda = 1, 2$, be arbitrary complex-valued functions on $\mathbb{F}_2^{n_\lambda}$. Let V_λ be the complex vector space $(\mathbb{C}^2)^{\otimes n_\lambda}$ of dimension 2^{n_λ}. Then $g^{(\lambda)}$ has a natural interpretation as a vector in V_λ. Here the basis vectors of V_λ are labelled by the elements of $\mathbb{F}_2^{n_\lambda}$. Any function $h : \mathbb{F}_2^{n_1+n_2} \rightarrow \mathbb{C}$ can be considered as tensor in the space $V_1 \otimes V_2$. Then the coordinate of that tensor with respect to the basis vector labeled by $b_1 \otimes b_2$, $b_\lambda \in \mathbb{F}_2^{n_\lambda}$, is equal to $h(b_1, b_2)$.

For $\lambda = 1, 2$ let $g^{(\lambda)} : \mathbb{F}_2^{n_\lambda} \rightarrow \mathbb{C}$ be as above. For $c \leq j \leq \min(n_1, n_2)$ we define a mapping $(g^{(1)} \odot g^{(2)})_{j,c} : \mathbb{F}_2^{n_1+n_2-j-c} \rightarrow \mathbb{C}$ by

$$\left(g^{(1)} \odot g^{(2)}\right)_{j,c} (x', x_1, x_2) = \sum_{x \in \mathbb{F}_2^c} g^{(1)}(x, x', x_1) \cdot g^{(2)}(x, x', x_2) \; ,$$

where $x' \in \mathbb{F}_2^{j-c}, \; x_\lambda \in \mathbb{F}_2^{n_\lambda - j}$. We abbreviate $(g^{(1)} \odot g^{(2)})_{j,0}$ to $(g^{(1)} \odot g^{(2)})_0$.

If $g^{(1)}$ and $g^{(2)}$ are quadratic mappings then $(g^{(1)} \odot g^{(2)})_n$ is a quadratic mapping by Lemma 1. Considering the corresponding quadratic state vectors we see that $(g^{(1)} \odot g^{(2)})_0$ is just the tensor product $g^{(1)} \otimes g^{(2)}$ of $g^{(1)}$ and $g^{(2)}$.

The functions $g^{(\lambda)}, \lambda = 1, 2$ defined above may also be considered as tensors in the spaces $V_0 \otimes V_\lambda$ with $V_0 = (\mathbb{C}^2)^{\otimes j}$, $V_\lambda = (\mathbb{C}^2)^{\otimes n_\lambda - j}$. Then $g^{(1)} \otimes g^{(2)} \in V_0 \otimes V_1 \otimes V_0 \otimes V_2$. We assume that V_0 is equal to it dual space via the standard Euclidean scalar product. Then we obtain the contraction of $g^{(1)} \otimes g^{(2)}$ over the two copies of V_0 as $(g^{(1)} \otimes g^{(2)})_{j,j}$. The result has to be interpreted as a tensor in the space $V_1 \otimes V_2$.

The tensors $g^{(\lambda)}$ in the space $V_0 \otimes V^\lambda$ given above can also be considered as $2^j \times 2^{k_\lambda}$ matrices M_λ, with $k_\lambda = n_\lambda - j$. Then the contraction $(g^{(1)} \otimes g^{(2)})_{j,j}$ given above corresponds to the matrix product $M_1^\top \cdot M_2$, which is a $k_1 \times k_2$ matrix.

In the next two sections we present an algorithm for computing $(g^{(1)} \odot g^{(2)})_{j,c}$ for quadratic mappings $g^{(1)}, g^{(2)}$. This yields an algorithm for tensor contraction and also for matrix multiplication of quadratic state matrices. Note that the transposition of a quadratic state matrix M is a rather simple operation that can be achieved by exchanging columns in the A part of a representation (e, A, Q) of M. The functions `qstate12_product` and `qstate12_matmul` in module `qmatrix12.c` implement the operation $(. \odot .)_{j,c}$ and the matrix multiplication.

The operator $(. \odot .)_{j,c}$ can be implemented in python with the `numpy` package. Let `c1` and `c2` be one-dimensional complex `numpy` arrays of length 2^{n_1} and 2^{n_2} corresponding to the vectors $g^{(1)}$ and $g^{(2)}$, respectively. Then a `numpy` array `c3` corresponding to the vector $(g^{(1)} \otimes g^{(2)})_{j,c}$ can be computed as follows:

```python
import numpy as np
c1a = c1.reshape((2**c, 2**(j-c), -1))
c2a = c2.reshape((2**c, 2**(j-c), -1))
c3 = np.einsum("cjk,cjl->jkl", c1a, c2a)
c3 = c3.reshape((-1,))
```

Without going into details, we remark that in the graphical ZX-calculus (which is used for describing linear maps between qubits) is an appropriate setup for 'explaining' the operation $(.\odot.)_{j,c}$, see e.g. https://en.wikipedia.org/wiki/ZX-calculus .

An algorithm for multiplying quadratic mappings

In this section we present an effective algorithm for computing the product $g^{(1)} \cdot g^{(2)}$ of two quadratic mappings $g^{(1)}, g^{(2)} : \mathbb{F}_2^n \rightarrow \mathbb{C}$. Such an algorithm is a key ingredient for implementing the operator $(.\odot.)_{j,c}$. For $\lambda = 1, 2$ let $(e^{(\lambda)}, A^{(\lambda)}, Q^{(\lambda)})$ be a reduced representation of $g^{(\lambda)}$.

We will show that for any $j \leq n$ there are quadratic mappings $g^{(\lambda,j)} = f\big(e^{(\lambda,j)}, A^{(\lambda,j)}, Q^{(\lambda,j)}\big)$ with $g^{(1,j)} \cdot g^{(2,j)} = g^{(1)} \cdot g^{(2)}$, where the first j columns of $A^{(1,j)}$ and $A^{(2,j)}$ are equal, and both, $A^{(1,j)}$ and $A^{(2,j)}$, are in reduced echelon form. We put $(e^{(\lambda,0)}, A^{(\lambda,0)}, Q^{(\lambda,0)}) = (e^{(\lambda)}, A^{(\lambda)}, Q^{(\lambda)})$.

Assume that $e^{(\lambda,j-1)}, A^{(\lambda,j-1)}, Q^{(\lambda,j-1)}$ satisfy the conditions given above. Then we can compute $\big(e^{(\lambda,j)}, A^{(\lambda,j)}, Q^{(\lambda,j)}\big)$ from $\big(e^{(\lambda,j-1)}, A^{(\lambda,j-1)}, Q^{(\lambda,j-1)}\big)$ as follows:

Case 1: Both, $A^{(1,j-1)}$ and $A^{(2,j-1)}$, have a row with leading coefficient in column j.

Since $A^{(1,j-1)}$ and $A^{(2,j-1)}$ are in reduced echelon form and the first $j-1$ columns of these two matrices are equal, we conclude that the first j columns of $A^{(1,j-1)}$ and $A^{(2,j-1)}$ are equal.

So we may put $\big(e^{(\lambda,j)}, A^{(\lambda,j)}, Q^{(\lambda,j)}\big) = \big(e^{(\lambda,j-1)}, A^{(\lambda,j-1)}, Q^{(\lambda,j-1)}\big)$ for $\lambda = 1, 2$.

Case 2: Only $A^{(1,j-1)}$ has a row with leading coefficient in column j.

Assume that this row of $A^{(1,j-1)}$ has index i. We add row i to all rows k of $A^{(1,j-1)}$ where $A^{(1,j-1)}_{k,j} \neq A^{(2,j-1)}_{k,j}$. Therefore we apply the operation $T_{i,k}$ defined in section *Changing the representation of a quadratic mapping*. So we obtain a representation $f\big(e^{(1,j-1)'}, A^{(1,j-1)'}, Q^{(1,j-1)'}\big)$ of $g^{(1,j-1)}$, where the first j columns of $A^{(1,j-1)'}$ and $A^{(2,j-1)}$ are equal, except for the coefficient in row i, column j.

We obtain $A^{(1,j)}$ from $A^{(1,j-1)'}$ by deleting row i of $A^{(1,j-1)'}$, and $Q^{(1,j)}$ from $Q^{(1,j-1)'}$ by deleting row and column i. We put $e^{(1,j)} = e^{(1,j-1)'}$. We put $\big(e^{(2,j)}, A^{(2,j)}, Q^{(2,j)}\big) = \big(e^{(2,j-1)}, A^{(2,j-1)}, Q^{(2,j-1)}\big)$.

By construction, matrix $A^{(1,j)}$ is in reduced echelon from and the first j columns of $A^{(1,j)}$ and $A^{(2,j)}$ are equal.

It remains to show that deleting row i of matrix $A^{(1,j-1)'}$ does not change $g^{(1)} \cdot g^{(2)}$.

Let $D^{(\lambda)}$ be the submatrix of $A^{(\lambda,j-1)'}$ that consists of the first j columns of $A^{(1,j-1)'}$. For computing $g^{(1)} \cdot g^{(2)}$ we only have to consider rows of matrix $D^{(1)}$ that are linear combinations of rows of matrix $D^{(2)}$, excluding row 0 of both matrices. By construction, $D^{(1)}$ and $D^{(2)}$ are in reduced echelon form, row i of $D^{(1)}$ has its leading coefficient in column j, and in $D^{(2)}$ there is no row with leading coefficient in column j. Thus row i of $D^{(1)}$ is not a linear combination of the rows of $D^{(2)}$, ignoring row 0 of $D^{(2)}$.

Case 3: Only $A^{(2,j-1)}$ has a row with leading coefficient in column j.

This case is symmetric to case 2, exchanging the role of $A^{(1,j-1)}$ and $A^{(2,j-1)}$.

Case 4: Neither $A^{(1,j-1)}$ nor $A^{(2,j-1)}$ has a row with leading coefficient in column j.

Case 4.1: Columns j of $A^{(1,j-1)}$ and $A^{(2,j-1)}$ are equal.

Then we may proceed as in case 1.

Case 4.2: Column j of $A^{(1,j-1)}$ and $A^{(2,j-1)}$ are equal except in row 0.

Assume that $x^{(1)}$ is in the support of $g^{(1,j-1)}$, $x^{(2)}$ is in the support of $g^{(2,j-1)}$, and that the leftmost $j-1$ bits of $x^{(1)}$ and $x^{(2)}$ are equal. Then from the properties of $A^{(1,j-1)}$ and $A^{(2,j-1)}$ we conclude that $x^{(1)}$ and $x^{(2)}$ must differ in the bit at position j. Thus $g^{(1,j-1)} \cdot g^{(2,j-1)}$ is the constant function 0, and we may put $e^{(1,j)} = e^{(2,j)} = 0$.

Case 4.3: There is an $i > 0$ with $A^{(1,j-1)}_{i,j} \neq A^{(2,j-1)}_{i,j}$.

Let i be the highest row index such that $A^{(1,j-1)}_{i,j} \neq A^{(2,j-1)}_{i,j}$ holds.

For $\lambda = 1, 2$ we add row i to all rows k of $A^{(\lambda,j-1)}$ where $A^{(1,j-1)}_{k,j} \neq A^{(2,j-1)}_{k,j}$ by applying operations $T_{i,k}$.

So we obtain a representation $f\big(e^{(1,j-1)'}, A^{(1,j-1)'}, Q^{(1,j-1)'}\big)$ of $g^{(1,j-1)}$, where the first j columns of $A^{(1,j-1)'}$ and $A^{(2,j-1)}$ are equal, except for the coefficient in row i, column j.

For $j = 1, 2$ we obtain $A^{(\lambda,j)}$ from $A^{(\lambda,j-1)'}$ by deleting row i of $A^{(\lambda,j-1)'}$. We obtain $Q^{(\lambda,j)}$ from $Q^{(\lambda,j-1)'}$ by deleting row and column i. We put $e^{(\lambda,j)} = e^{(\lambda,j-1)'}$.

A similar argument as in case 2 shows that matrices $A^{(1,j)}$ and $A^{(2,j)}$ are as required and that deleting row i from $A^{(1,j-1')}$ and $A^{(2,j-1')}$ does not change $g^{(1)} \cdot g^{(2)}$.

Now we may compute the product of $g^{(1)}$ and $g^{(2)}$ as follows:

$$g^{(1)} \cdot g^{(2)} = g^{(1,n)} \cdot g^{(2,n)} = f\big(e^{(1,n)} \cdot e^{(2,n)}, A^{(1,n)}, Q^{(1,n)} \odot Q^{(2,n)}\big) \ .$$

If the symmetric $(m+1) \times (m+1)$ bit matrices $Q^{(\lambda)}$, $\lambda = 1, 2$, represent the quadratic functions $q^{(\lambda)} : \mathbb{F}_2^m \rightarrow \mathbb{T}$, then we define $Q^{(1)} \odot Q^{(2)}$ as the symmetric $(m+1) \times (m+1)$ bit matrix representing the quadratic function $q^{(1)} \cdot q^{(2)}$. The entries $Q^{(1\odot 2)}_{i,j}$ of $Q^{(1)} \odot Q^{(2)}$ can be computed as follows:

$$\begin{array}{lll}
Q^{(1\odot 2)}_{i,j} = Q^{(1)}_{i,j} + Q^{(2)}_{i,j} & \text{for} & i,j > 0 \ , \\
Q^{(1\odot 2)}_{i,0} = Q^{(1\odot 2)}_{0,i} = Q^{(1)}_{0,i} + Q^{(2)}_{0,i} + Q^{(1)}_{i,i} \cdot Q^{(2)}_{i,i} & \text{for} & i > 0 \ , \\
Q^{(1\odot 2)}_{0,0} = 0 \ , & &
\end{array}$$

The corrections in row and column 0 of $Q^{(1\odot 2)}$ are necessary, since the diagonal entries of $Q^{(1)}`and:math:`Q^{(2)}$ are to be interpreted modulo 4.

So our algorithm allows us to multiply quadratic mappings effectively.

Remark

In certain cases we have to compute $\big(e^{(\lambda,j)}, A^{(\lambda,j)}, Q^{(\lambda,j)}\big)$ from $\big(e^{(\lambda,j-1)}, A^{(\lambda,j-1)}, Q^{(\lambda,j-1)}\big)$; and we have the additional information that e.g. the factor g_1 of the product $g_1 \cdot g_2$ is a quadratic mapping that does not depend on qubit i. Then the part $A^{(1,j-1)}$ of the representation of $g^{(1,j-1)}$ always has a row i such that $A^{(1,j-1)}_{i,j}$ is the only nonzero entry in row i and in column j of $A^{(1,j-1)}$. Furthermore, row i and column j of $Q^{(1,j-1)}$ are zero in that case. Thus only cases 1 and 2 can occur in the above computation, and adding row i of $A^{(1,j-1)}$ to any other row in that matrix affects column j of $A^{(1,j-1)}$ only, and does not affect $Q^{(1,j-1)}$. In our implementation we make use of this simplification wherever appropriate.

Computing tensor and matrix products

In this section we explain how to compute the operator $(.\odot,)_{j,c}$. For $\lambda = 1, 2$ let $g^{(\lambda)} : \mathbb{F}_2^{n_\lambda}$ be quadratic mappings and $0 \leq c \leq j \leq \min(n_1, n_2)$. We first show how to compute a representation of $(g^{(1)} \odot g^{(2)})_j$ from the representations of $g^{(1)}$ and $g^{(2)}$.

For actually computing $(g^{(1)} \odot g^{(2)})_j$ we may extend the mapping $g^{(1)} : \mathbb{F}_2^j \times \mathbb{F}_2^{n_1-j} \rightarrow \mathbb{C}$ to a mapping $g^{(1')} : \mathbb{F}_2^j \times \mathbb{F}_2^{n_1-j} \times \mathbb{F}_2^{n_2-j} \rightarrow \mathbb{C}$, with $g^{(1')}$ not depending on the last factor $\mathbb{F}_2^{n_2-j}$, as described in one of the last sections. Similarly, we may extend $g^{(2)}$ to a mapping $g^{(2')} : \mathbb{F}_2^j \times \mathbb{F}_2^{n_1-j} \times \mathbb{F}_2^{n_2-j} \rightarrow \mathbb{C}$, with $g^{(2')}$ not depending on the factor $\mathbb{F}_2^{n_2-j}$ in the middle. Then we simply have $(g^{(1)} \odot g^{(2)})_j = g^{(1')} \cdot g^{(2')}$.

Using the techniques discussed in section *Extending a quadratic mapping* and in the last section we can compute a representation of $(g^{(1)} \odot g^{(2)})_j$ from representations of $g^{(1)}$ and $g^{(2)}$.

It remains to compute $(g^{(1)} \odot g^{(2)})_{j,c}$ from $(g^{(1)} \odot g^{(2)})_j$. We have

$$\left(g^{(1)} \odot g^{(2)}\right)_{j,c}(x) = \sum_{y \in \mathbb{F}_2^c} \left(g^{(1)} \odot g^{(2)}\right)_j(y, x) \quad \text{for} \quad x \in \mathbb{F}_2^{n_1+n_2-j-c} .$$

Let $h : \mathbb{F}_2^n \rightarrow \mathbb{C}$ be a quadratic mapping with representation (e, A, Q), and define $h_c : \mathbb{F}_2^{n-c} \rightarrow \mathbb{C}$ by $x \mapsto \sum_{y \in \mathbb{F}_2^c} h(y, x)$. Then h_c is a quadratic mapping with representation (e, A_c, Q), where A_c is obtained from A by deleting the leftmost c columns of A.

So we may easily compute $(g^{(1)} \odot g^{(2)})_{j,c}$ from $(g^{(1)} \odot g^{(2)})_j$.

Restricting a quadratic mapping

For any function $g : \mathbb{F}_2^n \rightarrow \mathbb{C}$ define $\hat{g}^{(j)} : \mathbb{F}_2^n \rightarrow \mathbb{C}$ by

$$\hat{g}^{(j)}(x_{n-1}, \ldots, x_j, \ldots, x_0) = \left\{ \begin{array}{ll} g(x_{n-1}, \ldots, x_j, \ldots, x_0) & \text{if} \quad x_j = 0 \\ 0 & \text{if} \quad \mathrm{x}_j = 1 \end{array} \right.$$

Then $\hat{g}^{(j)} = g \cdot \hat{\chi}^{(j)}$, where $\hat{\chi}^{(j)} : \mathbb{F}_2^n \rightarrow \mathbb{C}$ is a projection function given by $(x_{n-1}, \ldots, x_j, \ldots, x_0) \mapsto 1 - x_j$. It is easy to find a representation of the quadratic mapping $\hat{\chi}^{(j)}$ so that we can compute a representation of $\hat{g}^{(j)}$ from a representation of a quadratic mapping g. This computation is implemented in function `qstate12_restrict_zero` in module `qstate12.c`.

Function $\hat{g}^{(j)}$ corresponds to a certain kind of a restriction of the function g. Let g be a quadratic mapping and (e, A, Q) be a representation of $\hat{g}^{(j)}$. Let $V = \mathbb{F}_2^{n-1-j} \times \{0\} \times \mathbb{F}_2^j$. Let $g \mid_V$ be the restriction of g to V. Then (e, A_j, Q) is a representation of the restriction $g \mid_V$, where A_j is the matrix obtained from matrix A by deleting column j. Function `qstate12_restrict` in module `qstate12.c` computes a representation of $g \mid_V$ from a representation of g.

The restriction of a quadratic mapping discussed in this section can be used for describing a measurement of a stabilizer state on a quantum computer, see e.g. [AG04].

Quadratic state matrices

A *quadratic state matrix* S of shape (n_0, n_1) is an element of the tensor product $V_0 \otimes V_1$ with the basis vectors of V_k being indexed by $\mathbb{F}_2^{n_k}, k = 0, 1$, such that the coordinate function of S is a quadratic mapping. That coordinate function is a function $\mathbb{F}_2^{n_0} \times \mathbb{F}_2^{n_1} \rightarrow \mathbb{C}$.

Thus S corresponds to a complex $2^{n_0} \times 2^{n_1}$ matrix. We may implement S as a quadratic state vector of $n_0 + n_1$ qubits, augmented by an information about its shape (n_0, n_1) . We let the n_0 qubits with high indices correspond to the rows of S; and we let the n_1 qubits with low indices correspond to the columns of S.

In python we implement a *quadratic state matrix* as a instance of class `QStateMatrix` in module `mmgroup.structures.qs_matrix`. A matrix of shape $(0, n)$ corresponds to a row vector of dimension 2^n and a matrix of shape $(n, 0)$ corresponds to a column vector of dimension 2^n. We have seen above that the unitary quadratic state matrices of shape (n, n) form the Clifford group $\mathcal{X}_n$. We have also discussed fast algorithms for multiplication and inversion of such matrices. So class `QStateMatrix` supports fast computation in Clifford group $\mathcal{X}_n$ for $n \leq 12$, which is sufficient for computing in the subgroup $2^{1+24}.\mathrm{Co}_1$ of the monster group.

On the C level, a quadratic state matrix is implemented as a structure of type `qstate12_type` as defined in file `clifford12.h`. E.g. function `qstate12_matmul` in file `qmatrix12.c` multiplies two such quadratic state matrices.

For practical calculations with quadratic state matrices it is vital to keep the following correspondences in mind.

Let S be a $2^{n_0} \times 2^{n_1}$ quadratic state matrix. We put $n = n_0 + n_1$. We implement Let S as a quadratic state vector $V = f(e, A, Q)$, where A is a $m \times n$ bit matrix and Q is a symmetric $m \times m$ bit matrix Q. So entry $S[i_0, i_1]$ corresponds to entry $i_0 \cdot 2^{n_1} + i_1$ of that quadratic state vector V for $0 \leq i_0 < 2^{n_0}$ and $0 \leq i_1 < 2^{n_1}$. Also, entry i of the quadratic state vector V corresponds to a row vector of n bits containing the binary representation of i.

We concatenate matrices A and Q to a $m \times (n + m)$ bit matrix M with $M[i, j] = A[i, j]$ and $M[i, j + n] = Q[i, j]$. In C and in python we implement the bit matrix as an array of unsigned 64-bit integers, so that bit $M[i, j]$ corresponds to bit j (of valence 2^j) of entry i of that array.

Displaying quadratic state vectors and matrices

We use the *bra-ket* formalism to display quadratic state vectors. We display a column unit vector as a *ket*. E.g. `|1011>` means the column unit vector labelled by the bit vector $(1, 0, 1, 1)$ in $(\mathbb{C}^2)^{\otimes 4}$. This corresponds to a state where the qubits with indices $3, 2, 1, 0$ have values $1, 0, 1, 1$, respectively. Similarly, we display a row vector as a *bra*. So e.g. `<110|` means the row unit vector labelled by $(1, 1, 0)$. A $2^4 \times 2^3$ matrix with an entry one in the row labelled by $(1, 0, 1, 1)$ and the column labelled by $(1, 1, 0)$, and zeros elsewhere, is displayed as `|1011><110|`.

We also want to display certain linear combinations of such unit vectors or matrices. As we have seen earlier, the coordinate function of such a state vector in $(\mathbb{C}^2)^{\otimes n}$ is a function $\mathbb{F}_2^n \rightarrow \mathbb{C}$. We only consider state vectors where the support of the coordinate function is an affine subspace of $\mathbb{F}_2^n$, and where the nonzero coordinates are in the set $\{\pm 1, \pm\sqrt{-1}\}$, up to a global scalar factor. Furthermore, the nonzero coordinates must be given by a certain quadratic form Q that we will specify below.

The support of the coordinate function is an affine subspace $a_0 + V$, where $a_0 \in \mathbb{F}_2^n$ and V is a linear subspace of $\mathbb{F}_2^n$ with a basis, say, $(a_1, \ldots a_m)$. In order to specify the support of the coordinate function, we write a_0 in *bra-ket* notation as above, and we write coordinates $a_1, \ldots, a_m$ underneath the coordinate a_0 without any bras or kets. We may give the basis $(a_1, \ldots, a_m)$ in echelon form, omitting leading zeros. We write A for the with row vectors $(a_0, \ldots, a_m)^\top$.

We write a lower triangular matrix Q with diagonal entries `'.'` or `'j'` and off-diagonal entries entries `'+'` or `'-'` to the left of the coordinate matrix A. Here an entry `'-'` means -1, `'j'` means $\sqrt{-1}$, and anything else means 1. We write $q_{i,j}$ for the entries of the matrix matrix Q. Then we put:

$$f(A, Q) = \sum_{x=(x_0,\ldots,x_m)\in\{0,1\}^{m+1}, x_0=1} Q(x) \cdot |\oplus_{k=0}^m x_k \cdot a_k\rangle \ , \quad Q(x) = \prod_{i,j=0}^m q_{i,j}^{x_i \cdot x_j} \ .$$

We use the same notation if the values a_i are *kets* or matrices written in the form `|ket><bra|`. For example:

```
.   |10><01|
-.    1  10
-+j       1
```

means $1 \cdot |10\rangle\,\langle 01| - 1 \cdot |11\rangle\,\langle 11| - \sqrt{-1} \cdot |10\rangle\,\langle 00| + \sqrt{-1} \cdot |11\rangle\,\langle 10|$.

E.g. the coefficient in the third term of that sum is computed as:

$$Q(1, 0, 1) = (1, 0, 1) \begin{pmatrix} . & & \\ - & . & \\ - & + & j \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} = (-1) \cdot \sqrt{-1} = -\sqrt{-1} \ .$$

It turns out that up to a scalar factor all elements of the complex Clifford group $\mathcal{X}_n$ and all stabilizer states can be displayed in that way. So we can display any element of $\mathcal{X}_n$ and any stabilizer state of $2n$ qubits on one page for n up to about 25.

Reducing a quadratic state matrix

We define a special *reduced matrix representation* (e, A, Q) of a quadratic state matrix S of shape (n_0, n_1) that differs slightly from the reduced representation of a quadratic state vector. That representation satisfies the following conditions:

1. We require that the representation (e, A, Q) is echelonized (but not necessarily reduced), as described in section *Reducing the representation of a quadratic mapping*.
2. Let A' be the bit matrix obtained from A by removing the leftmost n_0 columns from A. We require that a permutation of the rows of matrix A', excluding row 0, is in (not necessarily reduced) echelon form.

   Note that the first n_0 columns of the bit matrix A correspond to the 2^{n_0} rows of the complex matrix S; and that the remaining n_1 columns of A correspond to the 2^{n_1} columns of S.
3. Let K_0 be the set of the rows of the bit matrix A such that all bits in the leftmost n_0 columns of that row are zero. Let K_1 be the set of the rows of A such that all bits in rightmost n_1 columns of that row are zero. We exclude row 0 from K_0 and from K_1.

   If the bit matrix A has m' rows then the bit matrix Q is a symmetric $m' \times m'$ bit matrix. Let Q' be the submatrix of Q that consists of all rows with index in K_1 and of all columns with index in K_0 .

   We require that submatrix Q' of Q has at most one nonzero entry in each row and in each column.

   Note that $K_0 \cap K_1 = \emptyset$, since the representation (e, A, Q) is echelonized.

We may obtain a *reduced matrix representation* (e, A, Q) of a quadratic state matrix S of shape (n_0, n_1) as follows:

- Starting from a reduced representation (e_0, A_0, Q_0) of a quadratic state vector S we may obtain a representation (e_1, A_1, Q_1) of the matrix S satisfying condition *(2.)* by applying a sequence of transformations $T_{i,j}, i < j$. Then A_1 is also in echelon form. $T_{i,j}$ is defined in section *Implementation of quadratic mappings*.
- We apply a sequence of transformations $T_{i,j}, i < j$, with $i, j \in K_0$ or $i, j \in K_1$ to (e_1, A_1, Q_1). These transformations preserve the echelon form of A_1 and also property *(2.)*. Since $K_0 \cap K_1 = \emptyset$, these transformations act as row and column operations on the submatrix Q' of Q. So we may achieve property *(3.)* by a sequence of such transformations, thus obtaining a suitable representation (e, A, Q) of S.

Using a *reduced matrix representation* (e, A, Q) of a quadratic state matrix S has a variety of advantages.

- We can easily compute the rank of S as follows:

  For (e, A, Q) let K_0, K_1, Q' be defined as above. Let K_2 be the subset K_1 containing all rows of A such that the corresponding row of bit matrix Q' is zero. Then the binary logarithm of the rank of S is equal to the number of rows of matrix A with are neither in K_0 nor in K_2. Here we have to exclude row 0 of A.

  We omit the proof of this fact since we do not need it for our purposes.
- That representation can be used for decomposing S into a product $M_1 \cdot H \cdot M_2$ of quadratic state matrices, where M_1, M_2 are monomial and H is a Hadamard-like matrix. By a Hadamard-like matrix we mean a tensor product of 2×2 unit matrices and matrices $\left(\begin{smallmatrix} 1 & 1 \\ 1 & -1 \end{smallmatrix}\right)$.

  If S has shape (n, n) then such a decomposition reduces the complexity of multiplying S with an arbitrary complex vector form $O(4^n)$ to $O(n \cdot 2^n)$.

  Essentially, we have used such a decomposition for multiplying the non-monomial part of generator ξ of the monster $\mathbb{M}$ with a vector of our representation of $\mathbb{M}$. Since this special case is discussed elsewhere, we do not go into details here.

- In the next section we will introduce the Pauli group $\mathcal{P}_n$, which is an important normal subgroup of $\mathcal{X}_n$. Using the reduced matrix representation of an element S of $\mathcal{X}_n$ we can conjugate any element of $\mathcal{P}_n$ with S in $O(n^2)$ bit operations.

  With quadratic state matrices, a general matrix multiplication in the Clifford group $\mathcal{X}_n$ costs $O(n^3)$ bit operations.

Function `qstate12_reduce_matrix` in file `qmatrix12.c` converts any representation of a quadratic state matrix to a reduced matrix representation. For the sake of efficiency, we relax condition 3 on matrix Q' as follows. A permutation of the rows of Q' is in echelon form when the columns of Q' are reversed.

The Pauli group

The *Pauli group* $\mathcal{P}_n$ of n qubits is the normal subgroup of the Clifford group $\mathcal{X}_n$ generated by the not gates, the phase π gates in $\mathcal{X}_n$, and by the scalar multiples of the unit matrix by a fourth root of unity. It has structure $\frac{1}{2}(2_+^{1+2n} \times Z_4)$, exponent 4 and order 2^{2n+2}.

We represent an element of $\mathcal{P}_n$ as a product of $2n+2$ generators. Each generator may have exponent 0 or 1. The sequence of these exponents are stored as a bit vector as follows:

- Bit $2n+1$ corresponds to multiplication with the scalar $\sqrt{-1}$.
- Bit $2n$ corresponds to multiplication with the scalar -1.
- Bit $n+i$ with $0 \leq i < n$ corresponds to a not gate applied to qubit i.
- Bit i with $0 \leq i < n$ corresponds to a phase π gate applied to qubit i.

Factors are ordered by bit positions, with the most significant bit position occurring first. In the C language we represent bit vectors a integers as usual.

All generators commute and have order 2 except for the following cases:

- A phase π gate anticommutes with a not gate applied to the same qubit, i.e their commutator is the scalar -1.
- Of course, the scalar $\sqrt{-1}$ squares to the scalar -1.

Functions `qstate12_pauli_vector_mul` and `qstate12_pauli_vector_exp` in module `qs_matrix.c` perform multiplication and exponentiation in the Pauli group $\mathcal{P}_n$.

Given an element p of the Pauli group and a *reduced matrix representation* (e, A, Q) of a unitary quadratic state matrix S, we can quickly compute the conjugate $S \cdot p \cdot S^{-1}$ as follows:

- Right multiply S with p by applying the appropriate gates to S. This affects row 0 of the bit matrix A and row and column 0 of the symmetric bit matrix Q only.
- Restore the original values $A[0, j]$ for all $j < n$ and the original values $Q[0, k] = Q[k, 0]$ for all $k > n$ by applying appropriate transformations $T_{0,j}$ to the modified representation (e, A, Q). This does not change the value $S \cdot p$ of the complex matrix computed in the previous step. $T_{0,j}$ is defined in section *Implementation of quadratic mappings*.
- We may restore the the remaining original values of row and column 0 of Q by applying phase π gates to S. These gate operations correspond to a left multiplication with a element p_1 of the Pauli group.
- We may restore the the remaining original values of row 0 of A by applying not gates to S. These gate operations correspond to a left multiplication with a element p_2 of the Pauli group.
- Up to a known scalar factor we have obtained an equation $S = p_2 \cdot p_1 \cdot S \cdot p$, with p_1, p_2 in the Pauli group. With this equation the requested conjugation is an easy computation in the Pauli group.

Function `qstate12_pauli_conjugate` in module `qs_matrix.c` performs this conjugation.

This operation can be simplified considerably if we require the product $S \cdot p \cdot S^{-1}$ up to a scalar factor only. Then the mapping $p \mapsto S \cdot p \cdot S^{-1}$ is a symplectic linear transformation on the vector space $\mathcal{P}_n / Z(\mathcal{P}_n)$ of dimension $2n$ over

$\mathbb{F}_2$. Here $Z(\mathcal{P}_n)$ is the centre of $\mathcal{P}_n$. Function `qstate12_to_symplectic` in module `qs_matrix.c` computes this transformation as a $2n \times 2n$ bit matrix acting on $\mathbb{F}_2^{2n}$ by right multiplication.

This means that the homomorphism of the Clifford group $\mathcal{C}_n$ group of structure $\frac{1}{2}(2_+^{1+2n} \times Z_8).\mathrm{Sp}_{2n}(2)$ onto the symplectic group $\mathrm{Sp}_{2n}(2)$ can be computed very fast.

Using these ideas and a suitable representation of the subgroup G_{x0} of the monster, we can also compute the homomorphism from the group G_{x0} of structure $2_+^{1+2n}.\mathrm{Co}_1$ onto its factor group Co_1 very fast. Here the image in Co_1 will be given as a linear operation on the Leech lattice modulo 2.

Applying a gate to a quadratic mapping

In the theory of quantum computing we may apply so-called *gates* to a quadratic state vector in $(\mathbb{C}^2)^{\otimes n}$. For our purposes a gate is a linear operation on $(\mathbb{C}^2)^{\otimes n} = (\mathbb{C}^2)^{\otimes k} \otimes (\mathbb{C}^2)^{\otimes n-k}$ which may be written as a tensor product of a unitary $2^k \times 2^k$ matrix *G* and a $2^{n-k} \times 2^{n-k}$ identity matrix for a small number $1 \leq k \leq 2$. Here we may permute the factors $\mathbb{C}^2$ of $(\mathbb{C}^2)^{\otimes n}$ arbitrarily before the decomposition into a tensor product as above.

We state without proofs how to apply a certain gates to a quadratic mapping. The gates listed below generate the Clifford group. Here a quadratic mapping represents a quadratic state vector as before. A quadratic state matrix of shape (n_0, n_1) is considered as a quadratic mapping $\mathbb{F}_2^{n_0} \times \mathbb{F}_2^{n_1} \rightarrow \mathbb{C}$.

The not gate

A *not* gate operating in qubit j maps a state g to a state g' with $g'(x) = g(x + e_j)$, where $e_j = (0, \ldots, 0, 1, 0, \ldots, 0)$ and the component 1 is at position *j*. A *not* gate operating in qubit j is implemented for $g = f(e, A, Q)$ by flipping the bit $A_{0,j}$. The C function `state12_gate_not` implements a not gate.

The controlled not gate

A *controlled not* gate is a gate that negates a target qubit $j \neq j'$ controlled by a qubit j'. Such a gate maps a state g to a state g' with $g'(x) = g(x + \langle e_{j'}, x\rangle \cdot e_j)$, where $\langle ., . \rangle$ is the scalar product of bit vectors. Such a gate is implemented for $g = f(e, A, Q)$ by adding column j' of A to column j of A. The C function `state12_gate_ctrl_not` implements a controlled not gate.

The phase ϕ gate

Applying a phase ϕ gate to qubit j of a state $g = f(e, A, Q)$ changes the state g to a state g' with

$$g'(x_{n-1}, \ldots, x_j, \ldots, x_0) = \exp(\phi x_j \sqrt{-1}) \cdot g(x_{n-1}, \ldots, x_j, \ldots, x_0) \ .$$

We consider only phases ϕ which are multiples of $\pi/2$. For an $(m + 1) \times n$ matrix A let A_j be the j-th column of matrix A. Let A_{-1} be the column vector $(1, 0, \ldots, 0)^\top$ with $m + 1$ entries. Then a phase π gate on qubit j maps $f(e, A, Q)$ to

$$f\left((-1)^{A_{0,j}} \cdot e, A, Q \odot A_{-1}A_j^\top \odot A_j A_{-1}^\top\right) \ .$$

A phase $\pi/2$ gate on qubit j maps $f(e, A, Q)$ to

$$f\left(\sqrt{-1}^{A_{0,j}} \cdot e, A, Q \odot A_j A_j^\top\right) \ .$$

Here we consider A_j, A_{-1} as a $(m + 1) \times 1$ matrices, so that the matrix product $A_j A_{-1}^\top$ is an $(m + 1) \times (m + 1)$ matrix. Operator $\odot$ is as in section *Multiplication of quadratic mappings*.

The C function `qstate12_gate_phi` implements phase gate.

The controlled phase π gate

Applying a controlled phase π gate to qubits j and j' of a state $g = f(e, A, Q)$ changes the state g to a state g' with

$$g'(\ldots, x_j, \ldots, x_{j'}, \ldots) = (-1)^{x_j x_{j'}} \cdot g(\ldots, x_j, \ldots, x_{j'}, \ldots) \; .$$

A controlled phase π gate on qubit j and j' maps $f(e, A, Q)$ to

$$f\left((-1)^{A_{0,j} \cdot A_{0,j'}} \cdot e, A, Q \odot A_j A_{j'}^\top \odot A_{j'} A_j^\top\right) \; .$$

The C function `qstate12_gate_ctrl_phi` implements controlled phase gate.

The Hadamard gate

A Hadamard gate at qubit j is a a mapping that changes a quadratic mapping g to another quadratic mapping $1/\sqrt{2} \cdot g'$ with

$$g'(\ldots, x_{j+1}, x_j, x_{j+1}, \ldots) = \\ g(\ldots, x_{j+1}, 0, x_{j-1}, \ldots) + (-1)^{x_j} \cdot g(\ldots, x_{j+1}, 1, x_{j-1}, \ldots) \; .$$

We implement the application of a Hadamard gate on qubit j to a quadratic mapping g represented as (e, A, Q) as follows.

We append a zero row at A and also a zero row and a zero column at Q. Let i be the index of the appended row and column. Then we change $Q_{i,k}$ and $Q_{k,i}$ to $A_{k,j}$, $A_{k,j}$ to 0 for all $k \neq i$, and $A_{i,j}$ to 1. Let A', Q' be the modified matrices A', Q'. Then we have $g' = f(e, A', Q')$.

The C function `state12_gate_h` implements a Hadamard gate.

Correctness of the implementation of the Hadamard gate

The correctness of that implementation can be seen as follows. W.l.o.g we assume that j is the last index 0. Let $x = (x_{n-1}, \ldots, x_0) \in \mathbb{F}_2^n$, $y = (y_0, \ldots, y_m) \in \mathbb{F}_2^{m+1}$ with $y_0 = 1$, and assume $y \cdot A = x$ for the matrix product $y \cdot A$. Then

$$(y, b) \cdot A' = (x_{n-1}, \ldots, x_1, b) \quad \text{for} \quad b \in \mathbb{F}_2 \; .$$

Let q, q' be the quadratic mappings given by Q, Q'. Then

$$q'(y, b) = (-1)^{b \cdot \langle y, A_0 \rangle} \cdot q(y) = (-1)^{b \cdot x_0} \cdot q(y) \; ,$$

where A_0 is the last column of A. Thus

$$f(e, A', Q')(x_{n-1}, \ldots, x_0) = g(x_{n-1}, \ldots, x_1, 0) + (-1)^{x_0} \cdot g(x_{n-1}, \ldots, x_1, 1) \; .$$

Computing the trace of a quadratic state matrix

We briefly explain the computation of the trace of a quadratic state matrix S of shape (n, n). Such a matrix is considered as a mapping $\mathbb{F}_2^n \times \mathbb{F}_2^n \rightarrow \mathbb{C}$. So there are n qubits corresponding to the rows and n qubits corresponding the columns of the matrix.

For $i = 0, \ldots, n-1$ we apply a control not gate to the i-th row qubit, controlled by the i-th column qubit. This moves the diagonal entries of the matrix S to row 0.

Using the techniques described above we may restrict the modified matrix to row 0, and sum up all entries of that row. This leads to a scalar that has the value of the trace of S.

1.6.3 Class `QStateMatrix` modelling a quadratic state matrix

class `mmgroup.structures.qs_matrix.`**`QStateMatrix`**(**args: Any*, ***kwargs: Any*)

This class models a quadratic state matrix

Quadratic state matrices are described in the *API reference* in section **Computation in the Clifford group**.

Parameters

- **`rows`** (`int` or and instance of class `QStateMatrix`) –
  - Binary logarithm of the number of rows of the matrix
  - or an instance of class `QStateMatrix`. Then a deep copy of that instance is created.
- **`cols`** (A nonnegative `int`, if parameter `rows` is an `int`) – Binary logarithm of the number of columns of the matrix
- **`data`** – The data of the matrix as described below
- **`mode`** – Evaluated according as described below

Raise

- TypeError if `type(data)` is not as expected.
- ValueError if `data` cannot be converted to an instance of class `QStateMatrix`.

In terms of the theory of quantum computing, `rows, cols = 0, n` creates a column vector or a *-ket* `|v>` corresponding to a state of of `n` qubits, and `rows, cols = n, 0` creates a row vector or a *-bra* `<v|` corresponding to a linear function on a state of `n` qubits.

If `rows == cols == n` and the created `2**n` times `2**n` matrix is invertible, then the matrix is (a scalar multiple of) an element of the complex Clifford $\mathcal{X}_{12}$ of `n` qubits described in [NRS01].

If `rows` is an instance of this class then a copy of that instance is created.

If `rows` and `cols` are integers then `data` may be:

- `None` (default). Then the zero matrix is created.
- A list of integers. Then that list of integers must encode a valid pair `(A, Q)` of bit matrices that make up a state, as described in *Long-term stable storage of vectors of the representation*, subsection **Quadratic state matrices**. In this case parameter `mode` is evaluated as follows:
  - 1: create matrix `Q` from lower triangular part
  - 2: create matrix `Q` from upper triangular part
  - Anything else: matrix `Q` must be symmetric.
- An integer `v`. Then one of the values `rows` and `cols` must be zero and the unit (row or column) vector with index `v` is created. Here a row vector is an `1` times `2**cols` and a column vector is a `2**rows` times `1` matrix.

As in `numpy`, matrix multiplication of quadratic state matrices is done with the `@` operator and elementwise multiplication of such matrices is done with the `*` operator. A quadratic state matrix may also be multiplied by a scalar. Here the scalar must be zero or of the form:

$$2^{e/2} \cdot w \,, \quad e \in \mathbb{Z}, \; w \in \mathbb{C}, \; w^8 = 1 \; .$$

Divison by such a scalar is legal.

A matrix of type `QStateMatrix` may be indexed with square brackets as in `numpy` in order to obtain entries, rows, columns or submatrices. Then a complex `numpy` array (or a complex number) is returned as in `numpy`. It is not possible to change the matrix via item assignment. So the easiest way to obtain a complex version of an instance `qs` of type `QStateMatrix` is to write `qs[:,:]`.

A row or column index has a natural interpretation as a bit vector. In the theory of quantum computation a bit of such a bit vector corresponds to a *qubit*. Let `qs` be a quadratic state matrix of shape `(m, n)` and

$$x = \sum_{k=0}^{m-1} 2^k x_k, \; y = \sum_{k=0}^{n-1} 2^k y_k, \; x_k, y_k \in \{0, 1\}.$$

Then `qs[x,y]` is the entry of matrix `qs` with row index corresponding to the bit vector $(x_{m-1}, \ldots, x_0)$ and column index corresponding to the bit vector $(y_{n-1}, \ldots, y_0)$.

Officially, we support matrices with `rows, cols <= 12` only. Methods of this class might work for slightly larger matrices. Any attempt to constuct a too large matrix raises ValueError.

property H

Return conjugate transposed matrix as in numpy

property T

Return transposed matrix as in numpy

conjugate()

Compute complex conjugate of the matrix

Returns
instance of class `QStateMatrix`.

copy()

Return a deep copy of the matrix

extend(*j*, *nqb*, *copy=True*)

Insert `nqb` qubits at position `j`. .

Let `qs` be the state of shape `(n0+n1)`, and let `n = n0 + n1`. We change ``qs` to the following state `qs'` depending on `n + nqb` qubits:

`qs'(x[n-1],...,x[j],y[nqb-1],...,y[0],x[j-1]...,x[0]) = qs(x[n-1],...,x[j], x[j-1]...,x[0])` .

The function returns *ket*, i.e. a column vector of shape `(n + nqb, 0)`.

If the input is reduced then the result is also reduced.

If parameter `copy` is True (default) then a copy of the matrix is modified and returned.

extend_zero(*j*, *nqb*, *copy=True*)

Insert `nqb` zero qubits at position `j`.

Let `qs` be the state of shape `(n0+n1)`, and let `n = n0 + n1`. We change ``qs` to the following state `qs'` depending on `n + nqb` qubits:

`qs'(x[n-1],...,x[j],y[nqb-1],...,y[0],x[j-1]...,x[0])` is equal to `qs(x[n-1],...,x[j], x[j-1]...,x[0])` if `y[0] = ... = y[nqb-1] = 0` and equal to zero otherwise.

The function returns *ket*, i.e. a column vector of shape `(n + nqb, 0)`.

If the input is reduced then the result is also reduced.

If parameter `copy` is True (default) then a copy of the matrix is modified and returned.

gate_ctrl_not(*vc*, *v*)

Apply controlled not gates to a state

Change the state `qs` to a state `qs'` with `qs'(x) = qs(x (+) <vc,x> * v)` where `'(+)'` is the bitwise xor operation, and `<.,.>` is the scalar product of bit vectors. The result is not reduced. The scalar product of the bit vectors `j` and `jc` must be zero. Otherwise the `ctrl not` operation is not unitary.

Computing `qs.gate_ctrl_not(1 << jc, 1 << j)`, for `jc != j`, corresponds to applying a controlled not gate to qubit `j` contolled by qubit `jc`. This operation is unitary if and only if the scalar product of `j` and `jc` is zero.

gate_ctrl_phi(*v1*, *v2*)

Apply controlled phase gates to a state

Change the state `qs` to a state `qs'` with `qs'(x) = qs(x) * (-1)**(<v1,x>*<v2,x>)`, where `<.,.>` is the scalar product of bit vectors and `'**'` denotes exponentiation. The result is reduced if the input is reduced. Computing `qs.gate_ctrl_phi(1 << j1, 1 << j2)` corresponds to applying a phase `pi` gate to qubit `j2` controlled by qubit `j1`.

gate_h(*v*)

Apply Hadamard gates to a state

Apply a Hadamard gate to all qubits `j` of the state `qs` (referred by `self`) with `v & (1 << j) == 1`. Aplying a Hadamard gate to gate `j` changes a state `qs` to a state `1/sqrt(2) * qs'`, where `qs'(..,x[j+1],x_j,x[j-1],..) = qs(..,x[j+1],0,x[j-1],..) + (-1)**(x_j) * qs(..,x[j+1],1,x[j-1],..)` . The result is not reduced.

gate_not(*v*)

Apply not gates to a state

Change the state `qs` to a state `qs'` with `qs'(x) = qs(x (+) v)`, where `'(+)'` is the bitwise xor operation. The result is reduced if the input is reduced. Computing `qs.gate_not(1 << j)` corresponds to negating qubit `j`.

gate_phi(*v*, *phi*)

Apply phase gate to a state

Change the state `qs` to a state `qs'` with `qs'(x) = qs(x) * sqrt(-1)**(phi * <v,x>)`, where `<.,.>` is the scalar product of bit vectors and `'**'` denotes exponentiation. The result is reduced if the input is reduced. Computing `qs.gate_phi(1 << j, phi)` corresponds to applying a phase `(phi * pi/2)` gate to qubit `j`.

inv()

Return inverse matrix

Returns an instance of class `QStateMatrix`. Raise ValueError if matrix is not invertible.

lb_norm2()

Return binary logarithm of squared operator norm.

The operator norm is the largest absolute singular value of the matrix. For a quadratic state matrix this is a power of $\sqrt{2}$ or zero.

The function returns -1 if the matrix is zero.

lb_rank()

Return binary logarithm of the rank of the matrix

The rank of a nonzero quadratic state matrix is a power of two. The function returns -1 if the matrix is zero.

order(*max_order*)

Try to find the order of a matrix

If the matrix `m` has order `e` for `e <= max_order`, i.e. `m.power(e)` is the unit matrix, then `e` is returned. Otherwise `ValueError` is raised.

The function might also succeed if `e` is slighty larger than `max_order`. It has run time `O(max_order**0.5)`.

pauli_conjugate(*v*, *arg=True*)

Conjugate Pauli group elements by the matrix

The method conjugates an element of the Pauli group or a list of such elements with the quadratic matrix and returns the congugated Pauli group element (or the list of conjugated elements). All Pauli group elements are encoded as in method `pauli_vector`.

If $M`isthequadraticstatematrixgivenbythisobjectthenthePauligroupelement:math:`v$ is mapped to MvM^{-1}.

Matrix M must be in a Clifford group. The function raises ValueError if this is not the case.

In case `arg = False` the (complex) arguments of the returned Pauli group elements are not computed.

pauli_vector()

Return matrix as a Pauli vector if it is in the Pauli group

We represent an element of the Pauli group $\mathcal{P}_n$ as a product of $2n+2$ generators. Each generator may have exponent 0 or 1. The sequence of these exponents are stored as a bit vector as follows:

- Bit $2n+1$ corresponds to multiplication with the scalar $\sqrt{-1}$.
- Bit $2n$ corresponds to multiplication with the scalar -1.
- Bit $n+i$ with $0 \leq i < n$ corresponds to a not gate applied to qubit i.
- Bit i with $0 \leq i < n$ corresponds to a phase π gate applied to qubit i.

Factors are ordered by bit positions, with the most significant bit position occuring first.

The function returns the bit vector corresponding to this object as an integer. It raises ValueError if the matrix iy not in the Pauli group.

power(*e*)

Return the `e`-th power of the matrix

For a matrix `m` the power with exponent `e = 2` is `m @ m`, and the power with `e = -1` is the inverse matrix of `m`.

reshape(*new_shape*, *copy=True*)

Reshape matrix to given shape

Parameters

- **new_shape** (*tuple of two integers*) – This shape of the reshaped matrix. It must be a pair of integers. A pair `(n0, n1)` correponds to a complex `2**n0` times `2**n1` matrix.
- **copy** (`bool`) – A deep copy of the reshaped matrix is returned if `copy` is True (default). Otherwise the matrix is reshaped in place.

`new_shape[0] + new_shape[1] = self.shape[0] + self.shape[0]` must hold.

If one of the values `new_shape[0]`, `new_shape[1]` is `-1` then the other value is calculated from the sum `self.shape[0] + self.shape[1]`.

`restrict`(*j*, *nqb*, *copy=True*)

This is method `restrict_zero` with deletion.

Let `qs` be the state of shape `(n0+n1)`, and let `n = n0 + n1`. We change ``qs` to the following state `qs'` depending on `n - nqv` qubits:

`qs'(x[n'-1],...,x[0])` is equal to `qs(x[n'-1],...,x[j],0,...,0,x[-1],...,x[0])`.

The function returns *ket*, i.e. a column vector of shape `(n - nqb, 0)`.

If the input is reduced then the result is also reduced.

If parameter `copy` is True (default) then a copy of the matrix is modified and returned.

`restrict_zero`(*j*, *nqb*, *copy=True*)

Restrict `nqb` qubits starting at postion `j` to `0`.

Let `qs` be the state of shape `(n0+n1)`, and let `n = n0 + n1`. We change ``qs` to the following state `qs'` depending on `n` qubits:

`qs'(x[n-1],...,x[0]) is equal to qs(x[n-1],...,x[0])` if `x[j] = ... = x[j+nqb-1] = 0` and equal to zero otherwise. We do not change the shape of `qs`.

The output is reduced if the input is reduced.

If parameter `copy` is True (default) then a copy of the matrix is modified and returned.

`rot_bits`(*rot*, *nrot*, *start=0*)

Rotate qubit indices of the state matrix `qs` in place

If the state matrix `qs` has shape `(0,n)` or `(n,0)` we rotate the qubits of `qs` in place as follows:

For `n0 <= i < n0 + nrot` we map qubit `i` to qubit `n0 + (i + rot) % nrot`. E.g. `nrot = 3, rot = 1, n0 = 0` means qubits are mapped as `0->1, 1->2, 2->0`.

Here the entries of the state matrix are labelled by bit vectors of length `n`. Let `qs(x[0],...,x[n-1])` be the entry of the state matrix corresponding to the bit vector `(x[n-1],...,x[0])`.

Then the function changes `qs to qs'` with `qs'(...,x[start+nrot],y[nrot-1],...,y[0], x[start-1],...) = qs(...,x[start+nrot],z[nrot-1],...,z[0],x[start-1],...)`, where `z[j] = y[j - rot (mod 3)]`.

If the state matrix `qs` has shape `(n0, n1)` then we label the entries of state matrix by bit vectors of length `n0 + n1`, with the `n1` highest bits corresponding to the rows of the matrix, and the `n0` lowest bits corresponding to the columns of the matrix.

`property shape`

Get shape of the complex matrix represented by the state

The function returns a pair `(rows, cols)` meaning that the state corresponds to a complex `2**nrows` times `2**ncols` matrix.

`show`(*reduced=True*)

Return a state as a string.

If `reduced` is True (default) then the reduced representation of the state is displayed. Otherwise the state is displayed 'as is'.

The standard conversion of a state to a string, i.e. method `__str__()`, is equivalent to method `show()`.

sumup(*j*, *nqb*, *copy=True*)

Sum up nqb qubits starting at position j.

Let qs be the state of shape (n0+n1), and let n = n0 + n1. We change qs to the following state qs' depending on n - nqb qubits:

qs'(x[n-1],...,x[j+nqb],x[j-1],...,x[0]) = sum_{x[j+nqb-1],...,x[j]} qs1(x[nn1-1],...,x[0]) .

The function returns *ket*, i.e. a column vector of shape (n - nqb, 0).

If the input is reduced then the result is also reduced.

If parameter copy is True (default) then a copy of the matrix is modified and returned.

to_symplectic()

Convert a quadratic state matrix to a symplectic bit matrix

Here the quadratic state matrix Q given by self must be an invertible $2^k \times 2^k$ quadratic state matrix, i.e. Q must be of shape (k,k). So Q is in the Clifford group $\mathcal{C}_k$ of k qubits, up to a scalar factor.

The function computes a $2k \times 2k$ bit matrix A with the following property.

If v is a bit vector representing an element g_v of the Pauli group of k qubits then $v \cdot A$ is the vector representing the element $Q \cdot g_v \cdot Q^{-1}$ of that Pauli group, up to a scalar factor. Here elements of the Pauli group are given as integers representing bit vectors in the same way as in method pauli_vector, ignoring the bits 2k, 2k+1 corresponding to the scalar factor.

So this method computes the natural homomorphism from the Clifford group $\mathcal{C}_k$ to the symplectic group $\mathrm{S}_{2k}(2)$.

The function returns A as a numpy array of shape (2*n), with the entries of that array corresponding to the rows of A.

trace()

Return the trace of a square matrix.

The trace is returned as an integer, a floating point number, or a complex number.

xch_bits(*sh*, *mask*)

Exchange qubit arguments the state qs in place

We label the entries of the state matrix qs by bit vectors and define qs(x[n-1],...,x[0]), with n = n0 + n1, (n0, n1) = qs.shape as in method rot_bits.

We exchange qubit j with argument qubit j + sh of the state, if bit j of mask is set. If bit j of mask is set then bit j + sh of mask must not be set. No mask bit at position >= n - sh may be set.

E.g. qs.qstate12_xch_bits(1, 0x11) changes the state matrix qs to a state matrix qs' with qs'(...,x6,x5,x4,x3,x2,x1,x0.) = qs(...,x6,x4,x5,x3,x2,x0,x1).

mmgroup.structures.qs_matrix.**qs_unit_matrix**(*nqb*)

Return unit matrix as an instance of class QStateMatrix

The returned unit matrix has shape (nqb, nqb). So it represents a 2**nqb times 2**nqb unit matrix.

1.6.4 Computation in the Subgroup G_x0 of the monster

According to Conway's construction of the monster $\mathbb{M}$ in [Con85], the subgroup G_{x0} of structure $2_+^{1+24}.\mathrm{Co}_1$ has a faithful rational representation on the tensor product $4096_x \otimes 24_x$. Here 24_x is the representation of the group Co_1 as the automorphism group of the real Leech lattice, and 4096_x is the representation of a group $G(4096_x)$ as a subgroup of the real Clifford group $\mathcal{C}_{12}$, where the representation of $\mathcal{C}_{12}$ is as in section *Class QStateMatrix modelling a quadratic state matrix*. Note that $G(4096_x)$ is also of structure $2_+^{1+24}.\mathrm{Co}_1$.

We remark that the subgroup G_{x0} of the monster has five classes of involutions that map to 2B involutions in the monster, see [Wil13]. From Table 2 in [Nor98] we conclude that G_{x0} has two classes of involutions that map to 2A involutions in the monster. According to [Con85], the restriction of the 196883-dimensional representation of the monster to G_{x0} leads to a representation of G_{x0} of shape

$$299_x \oplus 98280_x \oplus 24_x \otimes 4096_x \, .$$

We can compute the characters of all these representations of G_{x0}, e.g. with the C function `xsp2co1_traces_fast` in file `xsp2co1_traces.c`, or with function `xsp2co1_traces_all` in file `xsp2co1_elem.c`. The calculations in the python module `mmgroup.tests.test_involutions.make_involution_samples` show that the class of an involution in the group G_{x0} can be identified by computing the four characters given above.

We use the ideas in section *Class QStateMatrix modelling a quadratic state matrix* for implementing the representation 4096_x. In principle, the representation 24_x is straightforward. The part 4096_x of a representation of an element g of G_{x0} determines g up to sign. So g is uniquely determined if we store the image under g of a single vector in the representation 24_x. The space 24_x is equivalent to the Leech lattice Λ, and we store the image of a fixed shortest vector β in $\Lambda/3\Lambda$, which is the Leech lattice modulo 3, as follows.

Let ϕ be the natural homomorphism from the subgroup Q_{x0} of G_{x0} of structure 2^{1+24} onto $\Lambda/2\Lambda$, which is the Leech lattice modulo 2.

Let $g \in G_{x0}$ be represented as a pair $(g_1, g_2) \in G(4096_x) \times \mathrm{Co}_1$. Then g_2 can be reconstructed from the image $\beta \cdot g_2$ of a single shortest vector β in the Leech lattice (modulo 3).

The operation of g_2 on a shortest vector $\beta' \in \Lambda$ is given by

$$\beta' \mapsto \phi(g^{-1} \cdot \phi^{-1}(\beta' + 2\Lambda) \cdot g) \in \Lambda/2\Lambda,$$

up to sign. Here a short vector in $\Lambda/2\Lambda$ has precisely two opposite shortest preimages in Λ, and we may take an arbitrary preimage in Q_{x0} when applying the inverse ϕ^{-1} of ϕ.

We compute the image $\beta' \cdot g_2$ of an arbitrary shortest vector $\beta' \in \Lambda$ as follows. Assume first that β, β' are shortest vectors in Λ which are not perpendicular to each other, and assume that the shortest Leech lattice vector $\beta \cdot g_2$ is known for some $g_2 \in \mathrm{Co}_1$. If $\beta' \cdot g_2$ is known up to sign only, we can compute the sign of β' from the scalar products (β, β'), and $(\beta \cdot g_2, \beta' \cdot g_2)$, which must be equal. Since $|(\beta, \beta')| \in \{1, 2, 4\}$, it suffices to compute the scalar products modulo 3. For perpendicular shortest vectors β, β' we can easily find a shortest vector $\beta'' \in \Lambda$ with $(\beta, \beta'') \neq 0$ and $(\beta'', \beta') \neq 0$. So we may compute first $\beta'' \cdot g_2$ and then $\beta' \cdot g_2$ from these scalar products.

These ideas lead to a very fast implementation of the subgroup G_{x0} of the monster $\mathbb{M}$. Details of that implementation are given in the **C interface of the mmgroup project** in section *Computing in the subgroup G_{x0} of the Monster*.

Class `mmgroup.Xsp2_Co1` provides an implementation of the group G_{x0} based on these ideas.

1.6.5 Class `Xsp2_Co1` modelling an element of the group G_{x0}

class mmgroup.structures.xsp2_co1.**Xsp2_Co1**(*tag=None*, *atom=None*, **args*, ***kwds*)

Models an element the subgroup G_{x0} of the monster

Here the subgroup G_{x0} is a subgroup of the monster of structure $2^{1+24}.\mathrm{Co}_1$. It is generated by the elements $x_\delta, y_d, y_d, x_\pi, \xi$ described in section *The Monster group*.

The constructor of this class works exactly as the constructor of class *MM*. Here all elements of the monster $\mathbb{M}$ occuring in the constructor must lie in the subgroup G_{x0} of the monster. So in the constructor all tags are legal, except for the tag `'t'`. A instance of class *MM* is accepted in the constructor of this class and vice versa.

The group operation and the operation on a vector of class *MMVector* is the same as in class *MM*.

This class uses a considerably faster implementation of the group operation as in class *MM*, as described in section *Computation in the Subgroup G_x0 of the monster*.

The user hardly ever has to deal with this class, since the implementation of class *MM* uses the accelerated functions in this class automatically where appropriate.

chi_G_x0()

Compute characters of element

The function returns a tuple $(\chi_M, \chi_{299}, \chi_{24}, \chi_{4096})$ of integers.

Here χ_M is the character of the element in the 196833-dimensional rep 198883_x of the monster.

By Conway's construction of the monster we have:

$198883_x = 299_x \oplus 24_x \otimes 4096_x \oplus 98280_x$,

for suitable irreducible representations $299_x, 24_x, 4096_x, 98280_x$ of the group G_{x0}. The corresponding characters of the element of G_{x0} are returned in the tuple given above.

While the product $\chi_{24} \cdot \chi_{4096}$ is well defined, the factors χ_{24} and χ_{4096} are defined up to sign only. We normalize these factors such that the first nonzero value of the pair (χ_{24}, χ_{4096}) is positive.

conjugate_involution(*mmgroup=None*)

Find an element conjugating an involution standard element

If the element g given by `self` is an involution in the monster then the method computes an element h of the monster with $h^{-1}gh = z$, where z is define as follows:

If $g = 1$, we put $h = z = 1$

if g is a 2A involution (in the monster) then we let z be the involution in Q_{x0} corresponding to the Golay cocode word with entries $2, 3$ being set.

if g is a 2B involution (in the monster) then we let z be the central involution in G_{x0}

The function returns a pair `(I, h)`, where h as an element of the instance `MM` of class `MMGroup`. We put `I = 0` if $g = 1$. We put `I = 1, 2` if g is a 2A or 2B involution, respectively.

The function raises `ValueError` if g is not an involution.

This is a wrapper for the C function `xsp2co1_elem_conjugate_involution`.

Parameter `mmgroup` is not for public use. In special cases it may specify an alternative class implementing the monster group. Then the function returns h as an instance of that class.

conjugate_involution_G_x0(*guide=0*, *group=None*)

Map an involution in G_{x0} to a standard form.

Assume that the element g represented by `self` is an involution in the group G_{x0}.

The function computes an element a in G_{x0} such that $h = a^{-1}ga$ is a (fixed) representative of the class of g in the group G_{x0}. The the function returns a pair `(iclass, a)`, where `a` is the computed element of G_{x0}, and `iclass` describes the representative h of the involution class as given in the following list:

`iclass = '1A_x+'`: the neutral element x_1

`iclass = '2B_x-'`: the central involution x_{-1}

`iclass = '2A_x0'`: the element $x_{\{2,3\}}$

`iclass = '2B_x0'`: the element x_Ω

`iclass = '2A_o+'`: the element y_o

`iclass = '2B_o-'`: the element $x_{-1}y_o$

`iclass = '2B_o0'`: the element $x_{\{8,9\}}y_o$

`iclass = '2B_d0'`: the element $x_{\{0,12\}}y_d$

Here in $x_{\{i,j\}}$ the index $\{i,j\}$ indicates a Golay cocode word of length 2 given by the entries i and j. Octad o is the standard octad $\{0,1,2,3,4,5,6,7\}$. Dodecad d is the standard dodecad $\{0,4,8,13,14,15,17,18,19,21,22,23\}$.

The first two characters of the string `iclass` denote the class in the Monster containing that class. The last character of the string `iclass` denotes the sign of the character of the class in the representation $24_x \otimes 4096_x$.

By default, `a` is an instance of this class. If parameter `group` is set to a class representing a suitable subgroup of the monster (e.g. `group = mmgroup.MM`) then `a` is returned as an instance of that class.

Parameter `guide` should usually be zero. If `guide` is a type-4 vector v_4 in the Leech lattice mod 2 such that the two conditions $h = a^{-1}ga$ and $v_4 \cdot a = \Omega$ can both be achieved then we compute an element a satisfying these two conditions. Otherwise parameter `guide` is ignored. Here Ω is the standard frame in the Leech lattice.

`order`(*max_order=119*)

Return the order of the element of the monster group

If the argument `max_order` is present then the order of the element is checked up to (and including) `max_order` only. Then the function returns `0` if the order is greater than `max_order`. By default, the function returns the exact order of the element.

`property subtype`

Return subtype of an element

Let $g \in G_{x0}$ be stored in `self`. The function returns the subtype of a g. If g maps the standard frame Ω of the Leech lattice modulo 2 to a frame of subtype t then g has subtype t.

The subtype is returned as a pair of integers as in the corresponding method in class *`XLeech2`*, see section *Computations in the Leech lattice modulo 2* in the **guide** for background.

Since the subtype is determined by the size of the denominators of the representation 4096_x, it can be computed very fast.

`type_Q_x0`()

Return type of element if it is in the subgroup Q_{x0}

If the element is in the subgroup Q_{x0} of the monster then the function returns the type of the vector in the Leech lattice modulo 2 corresponding to this element. That type is 0, 2, 3, or 4.

The function raises ValueError if the element is not in the subgroup Q_{x0}.

1.7 The Coxeter group Y_{555} and the Bimonster

We want to find a homomorphism from the Coxeter group Y_{555} into the Bimonster

Module `mmgroup.bimm` contains an implementation of a (fixed) homomorphism from the Coxeter group Y_{555} into the Bimonster.

1.7.1 Description of the task to be done

The wreath product $\mathbb{M} \wr 2$ of the Monster $\mathbb{M}$ with the group Z_2 of order 2 is called the *Bimonster*. In ATLAS notation [CCN+85] the Bimonster has structure $(\mathbb{M} \times \mathbb{M}).2$. It is generated by $\mathbb{M} \times \mathbb{M}$ and an involution swapping the two copies of $\mathbb{M}$.

A Coxeter group is a group generated by a finite set $s_1, \ldots, s_k$ of involutions with a set of relations $(s_i, s_j)^{\alpha_{i,j}} = 1$ for each pair s_i, s_j of involutions. The Coxeter groups discussed in this application are given by graphs, where the vertices of the graph correspond to the generators s_i. In the sequel a vertex of a graph describing a Coxeter group will be called a *node*. If two nodes s_i, s_j are connected by an edge then we have a relation $(s_i, s_j)^3 = 1$; otherwise we have a relation $(s_i, s_j)^2 = 1$. In the last case the generators s_i and s_j commute.

Norton and Ivanov have shown that the Bimonster is isomorphic to a certain Coxeter group, together with a single additional relation, see [Nor92], [Iva92]. That presentation of the Bimonster is called Y_{555}.

The graph corresponding to the Coxeter relations in Y_{555} is given in the following Figure 1. The names of the generating reflections of Y_{555} in that figure are as in the ATLAS [CCN+85]. We let Y_{555} be the Coxeter group given by the

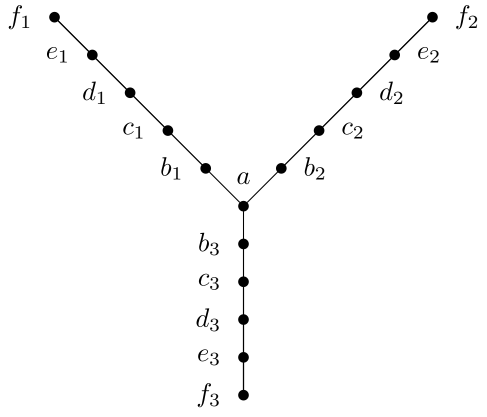

Fig. 1: Coxeter relations in the group Y_555

generators and relations in Figure 1 together with the following additional relation:

$$(ab_1c_1ab_2c_2ab_3c_3)^{10} = 1\,.$$

The purpose of this application is to implement the mapping from the group Y_{555} to the Bimonster. In the sequel we write Y_{555} for both, the graph in Figure 1, and the group defined above.

1.7.2 Extending Y_{555} to the incicdence graph of a projective plane

The graph Y_{555} can be extended to the incidence graph IncP3 of the projective plane P3 over the field $\mathbb{F}_3$. The 26-Node Theorem states that there is a homomorphism from the Coxeter group given by IncP3 to the Bimonster $\mathbb{M} \wr 2$, see [CNS88]. A presentation of the Bimonster with the generators of that Coxeter group and defining relations is well known see e.g. [Nor02], [Far12] for an overview. We also write IncP3 for the Coxeter group corresponding to to the incidence graph IncP3.

In practice it turns out that working with IncP3 is more flexible than working with Y_{555}.

The projective plane $P3$ contains 13 points P_i and 13 lines L_i, $0 \leq i < 13$. Point P_i is incident with line L_j if $i + j \equiv 0, 1, 3, \text{or } 9 \pmod{13}$. This construction of $P3$ is also used in [CNS88], [Nor02], and [Far12]. We may map the 16 nodes of Y_{555} to a subset of the set of the 26 nodes of IncP3 as follows:

a	b_1	b_2	b_3	c_1	c_2	c_3	d_1	d_2	d_3	e_1	e_2	e_3	f_1	f_2	f_3
P_0	L_0	L_1	L_3	P_1	P_2	P_{11}	L_8	L_7	L_5	P_5	P_7	P_4	L_4	L_6	L_{10}

In the description of the Monster group, the ATLAS [CCN+85] uses the names of the 16 nodes of Y_{555} given in the table above. It also uses names for remaining 10 nodes of $P3$, and it states the incidences between all these nodes. We number the remaining nodes in the ATLAS (preserving their incidence relations) as indicated in the follwing table:

f	a_1	a_2	a_3	g_1	g_2	g_3	z_1	z_2	z_3
L_9	P_3	P_{12}	P_{10}	P_9	P_8	P_6	L_{11}	L_2	L_{12}

1.7.3 Implementing the Bimonster and the homomorphism from Y_{555} to it

The projective plane over $\mathbb{F}_3$ and its automorphism group

The module implements the projective plane over $\mathbb{F}_3$

Module `inc_p3` implements the projective plane `P3` over $\mathbb{F}_3$ and its automorphism group. Points and lines in `P3` are implemented as instances of class `P3_node`, automorphisms of `P3` are implemented as instances of class `AutP3`.

We mainly work with the incidence graph containing the points and the lines of `P3` as vertices. The union of the set of the points and of the set of the lines in `P3` is called the set of the *nodes* of the projective plane `P3`.

class `mmgroup.bimm.`**`P3_node`**(*obj*)

Models a point or a line the projective plane P3.

We number the 13 points in the projective plane `P3` over $\mathbb{F}_3$ from 0 to 12, and the 13 lines from 13 to 25. Then a point with number `i` and a line with number `j` are incident if $i + j \equiv 0, 1, 3, \text{or } 9 \pmod{13}$.

Some strings are also accepted as a description of a point or a line in `P3`. The 13 points may be denoted as 'P0',…,'P12', and the the 13 lines may be denoted as 'L0',…,'L12'.

The names 'a', 'b1', 'b2', 'b3', 'c1', 'c2', 'c3', etc. refer to the embedding of the Y_{555} graph into the projective plane `P3`, as described in the documentation of the application. For background, see also [CNS88], [Far12].

Parameters

obj – An integer, a string, or an instance of class `AutP3` describing a point or a line in the projective plane `P3`.

`name()`

Return the name of the `P3` node in standard notation

property ord

Return internal number of instance of class `P3_node`

y_name()

Return the name of the `P3` node in Y_555 notation

class `mmgroup.bimm.`**AutP3**(*mapping=None*, *data=None*)

Models an automorphism of the projective plane `P3`.

This class models the automorphism group `AutP3` of the projective plane `P3` over the field $\mathbb{F}_3$.

Elements of `AutP3` should be given as (partial) mappings of points or lines. The standard way to describe an automorphism in `AutP3` is a dictionary containing a partial mapping of points or lines. Here the keys and the values of the dictionary must either all be points or all lines; they must be objects describing points or lines as in the constructor of class `AutP3`. A mapping between points or lines is accepted if it extends to a unique mapping of the projective plane `P3`.

Parameters

- **mapping** – Describes a mapping of points or lines in the projective plane `P3` as indicated in the table below.
- **data** – Additional data (optional) that describe a mapping of points or lines in some special cases.

Table 23: Legal types for parameter `mapping` in the constructor

type	Evaluates to
`None`	Creates the neutral element (default).
class `AutP3`	A deep copy of the given automorphism in class `AutP3` is returned.
`dict`	Dictionary containing a mapping between points or lines as described above.
`zip` object	`zip(x,y)` is equivalent to `dict(zip(x,y))`
`string` 'r'	Then we construct a random automorphism (depending on parameter `data`) as described below.
`string` 'p'	Then `data` must be a list of 13 integers (taken modulo 13), that describes a mapping of the 13 points.
`string` 'l'	Then `data` must be a list of 13 integers (taken modulo 13), that describes a mapping of the 13 lines.

Remarks:

If parameter `mapping` is the string `'r'`, then an optional parameter `data` of type `dict` or `zip` that describes a partial mapping of points or lines may follow. In this case we construct a random automorphism of `P3` satisfying the constraints of the mapping given by parameter `data`, if present. Such a random automorphism is chosen from a uniform distribution of all possible cases.

For instances `g1` and `g2` of this class, `g1 * g2` means group multiplication, and `g1 ** n` means exponentiation of `g1` with the integer `n`. `g1 ** (-1)` is the inverse of `g`. `g1 / g2` means `g1 * g2 ** (-1)`.

`g1 ** g2` means `g2**(-1) * g1 * g2`.

Multiplying an object of class `P3_node` with an object of class `AutP3` means application of an automorphism of `P3` to a point or line in `P3`.

line_map()

Return the automorphism as a line permutation list of length 13

Element `g` maps line `x` to line `g.line_map[x]`. The entries in the list are reduced modulo 13.

map()

Return the automorphism as a permutation list of length 26

Element `g` maps P3 node``x`` to node `g.map[x]`. Here the indices and values in the returned list are the numbers of the nodes as in class `P3_node`.

order()

Return order of element of the group AutP3

point_map()

Return the automorphism as a point permutation list of length 13

Element `g` maps P3 point``x`` to point `g.map[x]`.

`mmgroup.bimm.`**P3_incidences**(**x*)

Return list of P3 nodes incident with given P3 nodes

Here each argument of the function is a list of nodes of P3 describing a set S_i of nodes (i.e. points or lines) of P3. An entry of such a list may be anything that is accepted by the constructor of class `P3_node`. An integer argument is interpreted as a singleton, i.e. a set of size 1. A comma- separated string of names of P3 nodes is accepted as a set of P3 nodes.

The function returns the sorted list of P3 nodes (i.e instances of class `AutP3` that are incident with at least one node in each set S_i and not contained in any of the sets S_i.

`mmgroup.bimm.`**P3_incidence**(**x*)

Return (unique) P3 node incident with given P3 nodes

Here each argument describes a P3 node (i.e. a point or a line); it may be anything accepted by the constructor of class `P3_node`.

If there is a unique P3 node incident with all these P3 nodes then the function returns that node as an instance of class `P3_node`.

Otherwise the function raises ValueError.

This function is a simplified version of function `P3_incidences`. Its typical use case is to find a line through two points or the intersection point of two lines.

`mmgroup.bimm.`**P3_remaining_nodes**(*x1*, *x2*)

Complete points on a line or lines intersecting in a point

If arguments `x1, x2` are different points or lines in P3 then the function returns the list of the two remaining points on the line through `x1` and `x2`, or the list of the two remaining lines containing the intersection point of `x1` and `x2`, respectively. Otherwise the function raises ValueError.

Arguments `x1, x2` may by anything accepted by the constructor of class `P3_node`. The result is returned as list of two instances of class `P3_node`.

`mmgroup.bimm.`**P3_is_collinear**(*l*)

Check if list of P3 nodes contains 3 collinear nodes

Argument `l` of the function is a list of nodes of P3 describing a set of nodes (i.e. of points or lines) of P3. An entry of such a list may be anything that is accepted by the constructor of class `P3_node`. A comma-separated string of names of P3 nodes is accepted as a set of nodes.

The function returns `True` if that set of nodes contains 3 collinear points or 3 collinear lines, and `False` otherwise.

Norton’s presentation of the Monster group

This module implements Norton’s generators of the Monster.

Norton [Nor02] has given a presentation of the Monster group that greatly simplifies a mapping from the *projecive plane* presentation of the Monster to the representation of the Monster in [Gri82] and [Con85]. Our implemention of the Monster is based on the representation in [Con85]. So we may use the presentation in [Nor02] to construct a homomorphism from the *projecive plane* representation of the Monster into our implementation of the Monster that can be computed in practice.

`mmgroup.bimm.`**`Norton_generators`**`(`*check=False*`)`

Return the images of Norton’s generators in the Monster

Norton [Nor02] defines a presention of the Monster with generators (s, t, u, v, x) and relations.

The function returns the images of the presentation of these generators in the Monster under a (fixed) homomorphism. The result is returned as a quintuple of instances of class `MM`, corresponding to the images of these generators in the Monster.

If parameter `check` is `True` then we also check the relations in this presentation.

The Bimonster and its presentation Y_{555}

This module implements the Bimonster.

Class `BiMM` in this module implements an element of the Bimonster $\mathbb{M} \wr 2$ as described in the documentation of this application. Let IncP3 be the Coxeter group as in that documentation. Function `P3_BiMM` maps a word of generators of IncP3 into the Bimonster. The generators of IncP3 correspond to the points and lines of the projective plane P3 over the field $\mathbb{F}_3$. A point or a line of P3 is implemented as an instance of class `P3_node` in module `inc_p3`.

There is also a natural mapping from the automorphism group of P3 into the Bimonster compatible with the mapping from the Coxeter group into the Bimonster. Function `AutP3_BiMM` computes that mapping. An automorphism of P3 is implemented as an instance of class `AutP3` in module `inc_p3`. For background see [Nor02].

`class` `mmgroup.bimm.`**`BiMM`**`(`**args*`)`

This class models an element of the Bimonster.

The Bimonster is the group $\mathbb{M} \wr 2$ of structure $(\mathbb{M} \times \mathbb{M}).2$, where $\mathbb{M}$ is the Monster group.

Parameters

- **`m1`** (Instance of class `MM`) – An element m_1 of the Monster that embeds into the Bimonster as $(m_1, 1)$. Default is the neutral element 1 of the Monster.
- **`m2`** (Instance of class `MM`) – An element m_2 of the Monster that embeds into the Bimonster as $(1, m_2)$. Default is the neutral element 1 of the Monster.
- **`e`** (*`integer`*) – An optional exponent of the involution α, default is 0. Conjugation of an element of $\mathbb{M} \times \mathbb{M}$ by α means swapping the two factors in the direct product.

Returns

The element $(m_1, m_2) \cdot \alpha^e$ of the Bimonster

Return type

Instance of class `BiMM`

Arguments `m1` or `m2` may be anything that is accepted by the constructor of class `MM` as a single argument.

Alternatively, `m1` may be an instance of class `BiMM`; then arguments `m2` and `e` must be dropped.

Let `g1` and `g2` be instances of class `BiMM` representing elements of the Bimonster group. `g1 * g2` means group multiplication, and `g1 ** n` means exponentiation of `g1` with the integer `n`. `g1 ** (-1)` is the inverse of `g`. `g1 / g2` means `g1 * g2 ** (-1)`. We have `1 * g1 == g1 * 1 == g1` and `1 / g1 == g1 ** (-1)`.

`g1 ** g2` means `g2**(-1) * g1 * g2`.

`decompose``()`

Decompose the element of the Bimonster

The function returns a triple `(m1, m2, e)` such that the element of the Bimonster is equal to $(m_1, m_2) \cdot \alpha^e$. Here `m1` and `m2` are instances of class `MM` representing the elements m_1 and m_2 of the Monster, and `e` is equal to 0 or 1.

`order``()`

Return the order of the element of the Bimonster

`mmgroup.bimm.`**`P3_BiMM`**`(`*pl=[]*`)`

Map a word of generators in IncP3 into the Bimonster

Parameters

`pl` (List containing integers, strings, or instances of class `P3_node`) – List of generators in IncP3. Each entry in the list should be an instance of class `P3_node`. Such an entry may also be an integer or a string accepted by the constructor of class `P3_node`.

Returns

The image of the word of the given generators in the Bimonster

Return type

Instance of class `BiMM`

An integer `pl` or an instance `pl` of class `P3_node` is considered as a word of length 1.

`pl` may also be a string of alphanumric identifiers separated by commas. This is interpreted as a sequence of generators, where the names of the generators are interpreted as in the constructor of class `P3_node`.

`mmgroup.bimm.`**`AutP3_BiMM`**`(`*g*`)`

Map an automorphism of P3 into the Bimonster

Parameters

`g` (Instance of class `AutP3`) – Automorphism of P3

Returns

The image of the automorphism of P3 in the Bimonster

Return type

Instance of class `BiMM`

Parameters `g` may be anything that is accepted by the constructor of class `AutP3` as a single argument.

Example: Checking the spider relation in the group Y_{555}

As stated in the description of the group Y_{555} , the spider relation in that group is:

$$(ab_1c_1ab_2c_2ab_3c_3)^{10} = 1\,.$$

We can quickly check the spider relation as follows:

```python
# Class BiMM implements an element of the Bimonster.
# Function P3_BiMM maps a product of generators of
# the Coxeter group IncP3 to the Bimonster.
from mmgroup.bimm import  BiMM, P3_BiMM
# List of generators of IncP3 corresponding to the spider relation
spider = ['a', 'b1', 'c1', 'a', 'b2', 'c2', 'a', 'b3', 'c3'] * 10
# Let 'Spider' be the image of the spider relation in the Bimonster
Spider = P3_BiMM(spider)
# Check that this is equal to >the neutral element of the Bimonster
assert Spider == BiMM(1)
```

Alternatively, the spider relation can be verified as follows:

```python
from mmgroup.bimm import  P3_BiMM
# Put Spider1 = a * b_1 * c_1 * a * b_2 * c_2 * a * b_3 * c_3.
# Here we enter that product into the Bimonster as a string
Spider1 = P3_BiMM('a, b1, c1, a, b2, c2, a, b3, c3')
# Check that Spider1 has order 10 in the Bimonster
assert Spider1.order() == 10
```

1.7.4 Construction of the mapping from the Coxeter group into the Bimonster

Let IncP3 be the Coxeter group generated by the nodes $P_i, L_i, 0 \leq i < 13$ of the projective plane P3 as above, and let AutP3 be the automorphism group of P3. Let IncP3 : AutP3 be the split extension of the group IncP3 by the factor group AutP3, where AutP3 operates naturally on the generating reflections of the Coxeter group IncP3.

In this section we construct a mapping from IncP3 : AutP3 to the Bimonster.

Therefore we use the Norton's presentation [Nor02] of the Monster. Then we follow the ideas in Farooq's thesis [Far12] for extending that presentation of the Monster to a homomorphism from the Coxeter group IncP3 to the Bimonster. We assume that the reader is familiar with [Nor02] and we will adopt notation from ibid.

Norton's presentation of the Monster and the Bimonster

Norton [Nor02] defines a presentation (s, t, u, v, x, α) of the group IncP3 : AutP3, where $s, t, u \in$ AutP3, $v = \prod_{i=1}^{12} P_i$, $x = P_0 L_0$, $\alpha = P_0 v$. Then he adds a relation that maps IncP3 to the Bimonster $\mathbb{M} \wr 2$, thus obtaining a presentation of $(\mathbb{M} \wr 2)$: AutP3. Next he shows that this is actually a presentation of the direct product $(\mathbb{M} \wr 2) \times$ AutP3. Then he adds a set of relations that cancel the factor AutP3, leading to a presentation of the Bimonster with the generators (s, t, u, v, x, α). It turns out that all these generators, except for α, are in the subgroup $\mathbb{M} \times \mathbb{M}$ of index 2 of the Bimonster $\mathbb{M} \wr 2$. Finally, he gives a further set of relations in the generators (s, t, u, v, x) cancelling the second factor of the direct product $\mathbb{M} \times \mathbb{M}$, thus obtaining a presentation of the Monster $\mathbb{M}$. We remark that $P_i^u = P_{i-1}$ and $L_i^u = L_{i+1}$, with indices to be taken modulo 13, so that P_i and L_i can easily be computed from the generators (s, t, u, v, x, α).

Let IncP3$^+$ be the subgroup of IncP3 generated be the products of generators P_i, L_i of IncP3 with an even number of factors. Then IncP3$^+$ has index 2 in IncP3, and the presentation (s, t, u, v, x) together with the relations mentioned above defines a mapping ϕ from IncP3$^+$: AutP3 into the Monster.

Mapping Norton's presentation into our representation of the Monster

Essentially, our task is to construct an explicit mapping from the generators (s, t, u, v, x, α) of IncP3 : AutP3 into the Bimonster $\mathbb{M} \wr 2$. A first step for achieving this goal is to actually construct a mapping ϕ from the generators (s, t, u, v, x) of IncP3$^+$: AutP3 to $\mathbb{M}$.

In [Nor02] for each point P_i an element P_i^* of of the subgroup $\mathbb{M} \times \mathbb{M}$ of the Bimonster is defined. The elements P_i^* are called the *stars* (corresponding to the points P_i); and it is shown that AutP3 permutes the stars P_i^* in the same way as it permutes the corresponding points. Actually, the stars P_i^* are defined as words of generators P_i, L_i of even length, modulo the relations defining the Bimonster.

According to [Nor02] we have $\phi(x) = \tau$ for the generator x, where τ is the triality element of the Monster. There it is also shown that the stars and the (products of an even number of) points map to elements of the extraspecial subgroup Q_{x0} (of structure 2^{1+24}) of the Monster. In [Nor02] explicit images of the points and stars in Q_{x0} are constructed up to sign. More specifically, the images of these elements are given in the Leech lattice modulo 2, which is isomorphic to the quotient of Q_{x0} by its centre $\{1, x_{-1}\}$. It is easy to see that the signs of the images of the points and stars may be chosen arbitrarily, with the only restriction that the product $P_0^* P_1^* P_3^* P_9^*$ must be mapped to the element x_Ω of Q_{x0}, and not to its negative $x_{-\Omega}$. It can be shown that the tuple $(P_0 P_1, \ldots, P_0 P_{12}, P_1^*, \ldots, P_{12}^*)$, when taken modulo the centre of Q_{x0}, corresponds to a basis of the Leech lattice modulo 2. So for defining the mapping ϕ on the points and the stars it suffices to specify the signs of the images of the entries of that tuple. It turns out that everything works fine if we declare all these signs to be positive. This means that for any such image $x_d x_\delta \in Q_{x0}$, with $d \in \mathcal{P}, \delta \in C^*$, we choose d to be a 'positive' element of the Parker loop $\mathcal{P}$, according to our construction of $\mathcal{P}$.

With this choice the mapping ϕ is uniquely defined on the points and on the stars; and we shall see that it is also uniquely defined on all generators (s, t, u, v, x) of the presentation given in [Nor02]. The functions `PointP3` and `StarP3` in module `mmgroup.bimm.p3_to_mm` compute the mapping ϕ on the points and the stars, respectively.

The mapping ϕ is already defined on x and on v (since v is a product of points). Generators s, t, u, are defined as automorphisms in AutP3; so it suffices compute $\phi(a)$ for $a \in$ AutP3.

It is also shown in [Nor02] that the images of elements of AutP3 are in the centralizer G_{x0} (of structure 2^{1+24}.Co$_1$) of the element x_{-1}. Note that the images of the points and the stars generate the group Q_{x0}; so the action of any automorphism in AutP3 is determined (as an element of G_{x0}) by its action on the points and the stars, up to sign. The group AutP3 is the simple group $L_3(3)$ in ATLAS notation. Thus it is generated by its elements of odd order. For any $g \in G_{x0}$ at most one of the elements $g, x_{-1} \cdot g$ has odd order, so that the correct image of any element of AutP3 of odd order (and hence the image of the any element of AutP3) in G_{x0} is uniquely defined. So we can also construct the images of the generators (s, t, u). Function `Norton_generators` in module `mmgroup.bimm.p3_to_mm` returns the images of the generators (s, t, u, v, x) under our mapping ϕ.

Faaroq [Far12] had to perform strenuous calculations for actually computing ϕ on AutP3, since he had to use the representations of the groups G_{x0} and $\mathbb{M}$ available in 2012. With our construction of the Monster and of G_{x0} we are in a much more comfortable situation. Function `AutP3_MM` in module `mmgroup.bimm.p3_to_mm` quickly computes the mapping ϕ on AutP3. In the remainder of this subsection we discuss the operation of that function.

The relevant automorphsims $a \in$ AutP3 are given as mappings between the 13 points and the 13 stars, where the stars are transformed in the same way as corresponding points. Using our contruction of ϕ on the points and the stars, the image $\phi(a)$ can be given as an automorphism of Q_{x0} in G_{x0}, where G_{x0} operates on Q_{x0} by conjugation. Given such an automorphism γ on Q_{x0}, the C function `xsp2co1_elem_from_mapping` in file `xsp2co1_map.c` computes the corresponding element g of the group G_{x0} up to sign.

We close with some remarks on how the C function `xsp2co1_elem_from_mapping` works.

Class `Xsp2_Co1` in module `mmgroup.structures.xsp_co1` wraps fast C functions for computing in the group G_{x0}, so that computations in G_{x0} are easy. Given an automorphism γ of Q_{x0} as above, we can easily compute the image $\gamma(\tilde{\Omega})$, where $\tilde{\Omega}$ is the vector in the Leech lattice modulo 2 corresponding to the standard frame in the Leech lattice. The C function `gen_leech2_reduce_type4` in file `gen_leech_reduce.c` can compute an element $h_1 \in G_{x0}$ that maps $\gamma(\tilde{\Omega})$ to $\tilde{\Omega}$, see section *Computations in the Leech lattice modulo 2* in the *guide for developers* for mathematical background. So if $g \in G_{x0}$ corresponds to the automorphism γ then gh_1 corresponds to an automorphism γ_1 of Q_{x0}

fixing $\tilde{\Omega}$. The stabilizer of $\tilde{\Omega}$ in G_{x0} is a group N_{x0} of structure $2^{1+24}.2^{11}.M_{24}$. The automorphism γ_1 can easily be computed from γ and h_1. So it remains the considerably simpler task to compute an element of N_{x0} corresponding to the automorphism γ_1 of Q_{x0}. This computation is done as described in the documentation of the C function `xsp2co1_elem_from_mapping`.

From the Monster to the Bimonster

In this section we construct a mapping Φ from the group $\mbox{IncP3 : AutP3}$ to the Bimonster using the mapping ϕ : IncP3$^+$: AutP3 $\rightarrow \mathbb{M}$ defined in the last subsection.

The generator α in IncP3 $\setminus$ IncP3$^+$ is an involution that acts on IncP3$^+$ (and also on $\mbox{IncP3}^+$: AutP3) by conjugation. Hence by Theorem 3.1 in [CNS88] there is a mapping Φ from $\mbox{IncP3 : AutP3}$ to the Bimonster $\mathbb{M} \wr 2$ given by:

$$\Phi(y) = (\phi(y), \phi(y^\alpha)) \in \mathbb{M} \times \mathbb{M} \, , \quad \mbox{for} \quad y \in \mbox{ IncP3}^+ \mbox{ : AutP3},$$

and $\Phi(\alpha)$ is the involution in $\mathbb{M} \wr 2$ swapping the two copies of $\mathbb{M}$. For simplicity, we will also write α for the element $\Phi(\alpha)$ of $\mathbb{M} \wr 2$.

Since $\alpha = \prod_{i=0}^{12} P_i$ commutes with all points P_i and also with AutP3, we have:

$$\begin{aligned}\Phi(P_i) &= \alpha \cdot (\phi(\alpha \cdot P_i), \phi(\alpha \cdot P_i)) \, , \\ \Phi(a) &= (\phi(a), \phi(a)) \, , \quad \mbox{for } a \in \mbox{AutP3} \, .\end{aligned}$$

It remains to compute $\Phi(L_i)$. Here have to compute $\phi(L_i^\alpha)$. Therefore we will use the follwing fact:

It easy to show that the elements of the Bimonster $\mathbb{M} \wr 2$ that are conjugate to α are precisely the elements of shape $\alpha \cdot (m, m^{-1})$, $m \in \mathbb{M}$.

Since $\Phi(P_0) = \alpha \cdot (\phi(\alpha \cdot P_0), \phi(\alpha \cdot P_0))$ holds for the involution P_0, we conclude that $\Phi(P_0)$ is conjugate to α in $\mathbb{M} \wr 2$. In a Coxeter group all nodes connected by a path of edges are conjugate; so the nodes $\Phi(L_i)$ are also conjugate to α. Together with the fact stated above we obtain:

$$\Phi(L_i) = \alpha \cdot (\phi(\alpha \cdot L_i), \phi(\alpha \cdot L_i)^{-1}) \, .$$

Note that $\phi(\alpha \cdot L_0) = \phi(v) \cdot \tau$, where τ is the triality element in the Monster. Furthermore, we have

$$\phi(\alpha \cdot L_i) = \phi(\alpha \cdot L_0)^{\phi(u)^i} \, .$$

1.8 Version history

Table 24: List of releases

Version	Date	Description
0.0.1	2020-05-20	First release
0.0.2	2020-06-04	Order oracle added; bugfixes
0.0.3	2020-06-10	Bugfixes in code generator
0.0.4	2020-06-15	MSVC compiler is now supported
0.0.5	2021-08-02	Word shortening in monster implemented
0.0.6	2021-12-01	Group operation accelerated
0.0.7	2021-12-01	Bugfix in version generation
0.0.8	2022-07-12	Performance improved
0.0.9	2022-09-01	Performance improved
0.0.10	2022-10-11	Support for cibuildwheel added
0.0.11	2022-10-19	Bugfixes and macOS version added
0.0.12	2023-01-09	Support for the group Y_555 and the Bimonster
0.0.13	2023-10-27	Supports numbering elements of the Monster, and Python 3.12
0.0.14	2024-01-19	Demonstration code for reduction in the Monster added
1.0.0	2024-01-23	First official release (documentation updated)
1.0.1	2024-02-23	Support for exporting C functions from libraries
1.0.2	2024-02-27	Changes in build system regarding shared libraries

CHAPTER

TWO

THE MMGROUP GUIDE FOR DEVELOPERS

2.1 Introduction

This *guide for developers* explains some selected topics of the `mmgroup` project which are relevant for developers who want to make changes or bugfixes. It also explains some mathematical aspects of the implementation which are not covered by [Sey20].

The *guide* is far from complete and we hope that we can provide more information in future versions of it.

2.2 Directory structure

The directory is organized as follows:

- **`[Root directory]`**
  Contains setup.py and configuration files
  - **`docs`**
    - **`sources`**
      Source files for documentation with Sphinx
  - **`src`**
    - `mmgroup` The directory containing the package, and also extensions and shared libraries
      - `dev` Contains stuff needed by developers, including scripts and source file for generating C code automatically.
      - `generate_c` The code generator (for generating C code automatically)
      - `structures` Some basic structures used by the modules in `mmgroup`
      - `tests` Test scripts and some algebraic structures used for testing with `pytest`

Directory `src/mmgroup/dev` has the following subdirectories:

- `c_files` Contains automatically generated C code
- `clifford12` Scripts for generating C code related to the subgroup $2^{1+24}.\mathrm{Co}_1$ of the monster and also to a Clifford group resembling that subgroup.
- `generators` Scripts for generating tables related to the monomial operation of the element ξ of $\mathbb{M}$, and also for generating C code related to certain subgroups of $\mathbb{M}$.
- `hadamard` Scripts for generating C code related to Hadamard matrices
- `mat24` Scripts for generating C code related to the Mathieu group Mat_{24}

- `mm_basics` Scripts for generating C code related to the representation ρ_p of $\mathbb{M}$
- `mm_op` Scripts for generating C code related to the operation of $\mathbb{M}$ on ρ_p for specific values p
- `mm_reduce` Scripts for generating C code related to the word shortening algorithm and the computation of the order in $\mathbb{M}$.
- `pxd_files` Contains files with extension `.pyx, .pxd, .pxi` used by `Cython`, which are automatically generated or copied from a different source location

Directory `src/mmgroup/tests` has the following subdirectories:

- `groups` Slow python implementations of certain groups, including a preimage of $\mathbb{M}$, for testing.
- `spaces` Slow python implementations of certain vector spaces, including the space ρ_p, for testing.
- `test_xxx` Test scripts (to be executed with `pytest`) for testing module `xxx`.

2.3 Some mathematical aspects of the implementation

2.3.1 Implementing Automorphisms of the Parker loop

A standard automorphism π of the Parker loop $\mathcal{P}$ is implemented as an array of `12` integers of type `uint32_t`. The lower `13` bits of the i-th entry of that array contain the image of the i-th basis element $(b_i, 0)$ of the Parker loop, where b_i is the i-th basis vector of the Golay code $\mathcal{C}$. These images describe the automorphism π uniquely. Each element of $\mathcal{P}$ has a unique representation as a tuple $d = (d, \lambda)$, $d \in \mathcal{C}$, $\lambda \in \mathbb{F}_2$.

A key task is to compute the image $(d, \lambda) \cdot \pi$ of an element (d, λ) of the Parker loop under the automorphism π. By Lemma 4.1 in [Sey20] there is a quadratic form q_π with associated bilinear form θ_π on the Golay code satisfying

$$\begin{aligned} \theta_\pi &= \theta^\pi + \theta \;, \quad \text{where} \quad \theta^\pi(d, e) = \theta(d^\pi, e^\pi) \;, \\ & \qquad\qquad\qquad\qquad (d, \lambda)^\pi = (d^\pi, \lambda + q_\pi(d)) \;, \end{aligned}$$

for any $(d, \lambda) \in \mathcal{P}$. So the image $(d, \lambda) \cdot \pi$ can be computed if $q_\pi(d)$ can be computed. A functional value of $q_\pi(d)$ can easily be computed from the associated bilinear from θ_π' if the values $q_\pi(b_i)$ are known for all basis vectors b_i. The values $q_\pi(b_i)$ can be computed from the sign bits of the images if the basis vectors.

So it suffices to compute the bilinear form θ_π as a 12×12 bit matrix. The first term θ of θ_π does not depend on π and can be stored as a constant. The second term θ^π of that matrix is computed as follows:

Let c_i = $(b_i)^\pi$, and let C be the matrix with entries $c_{i,j}$ such that c_i = $\sum_j c_{i,j} b_j$. So C is just the matrix of the images of the basis vectors, ignoring the signs of the Parker loop.

Row i, column j of matrix θ^π is equal to $\theta(c_i, c_j)$. Since $\theta(c_i, c_j)$ is linear in its second argument, we have

$$\begin{aligned} \theta^\pi(c_i, c_j) &= \sum_k c_{j,k} \theta(c_i, b_k) \;, \\ \theta^\pi(c_i) &= \theta(c_i) \cdot C^\top \;. \end{aligned}$$

Here $\theta^\pi(c_i)$ is just the i-th row of the bit matrix θ^π. We also store the values $\theta(d)$ for all $d \in \mathcal{C}$ in a table. Thus the bilinear form θ_π can essentially be computed as a product of two 12×12 bit matrices.

We store the lower triangular submatrix of the bit matrix θ_π in bits `13, ..., 24` of the array representing the automorphism π of the Parker loop.

2.3.2 Implementing generators of the Monster group

The operation $y_f \cdot x_e \cdot x_\epsilon$

The operation $g = y_f \cdot x_e \cdot x_\epsilon$ on ρ_p is coded in function `mm_op<p>_xy` in the automatically generated C file `mm<p>_op_xy.c` for the modulus `p`. The C function performs that operation for arbitrary $e, f \in \mathcal{P}$ and $\epsilon \in \mathcal{C}^*$ in a single pass. That operation is monomial and the formula for it can easily be deduced from the information in [Sey20].

For $v \in \rho_p$ the C program calculates the components in $V_A, V_B, V_C \ldots$ of the result $v \cdot g$ in their natural order. For implementing such a program it is useful to have a formula for the operation of $g^{-1} = x_\epsilon \cdot x_{\bar{e}} \cdot y_{\bar{f}}$. In the sequel we state the operation of g^{-1} on ρ_p.

In this subsection we use the operator '$\oplus$' for the *programmer's multiplication* in the Parker loop $\mathcal{P}$, which we define by:

$$(\tilde{d_1}, \lambda_1) \oplus (\tilde{d_2}, \lambda_2) = (\tilde{d_2} + \tilde{d_2},\ \lambda_1 + \lambda_2)\ , \qquad \tilde{d_1}, \tilde{d_2} \in \mathcal{C},\ \lambda_1, \lambda_2 \in \mathbb{F}_2\ .$$

Then '$\oplus$' is a simple XOR operation on a computer; and the standard product $d_1 d_2$ in the Parker loop is given by $d_1 d_2 = (-1)^{\theta(d_1,d_2)} d_1 \oplus d_2$ for $d_1, d_2 \in \mathcal{P}$. Note that '$\oplus$' depends on the selected cocycle θ. We may store the cocycles $\theta(d)$ in a table, with one 12-bit entry representing the element $\theta(dZ(\mathcal{P}))$ of the Golay cocode for each of the 2048 cosets $dZ(\mathcal{P})$ of $Z(\mathcal{P}) = \{\pm 1, \pm \Omega\}$ in $\mathcal{P}$. So multiplication in $\mathcal{P}$ is easy.

Put $d^{[0]} = d^+, d^{[1]} = d^-$, and $X^+_{d,\delta} = X_{\Omega d,\delta}$ if $x_{\Omega d,\delta}$ is short, $X^+_{d,\delta} = X_{d,\delta}$ otherwise. Then

$$\begin{aligned}
X^+_{d,i} &\xrightarrow{g^{-1}} (-1)^{s_X} \cdot X^+_{d \oplus f, i}\,, \quad \text{with} \\
s_X &= P(f) + P(ef) + (|\epsilon| + 1) P(d) + P(def) + \langle e, i \rangle + \left\langle d, i^{|\epsilon|} \epsilon A(e,f) \theta(f) \right\rangle\ ; \\
(d^{[\tau]} \otimes_1 i) &\xrightarrow{g^{-1}} (-1)^{s_{YZ}} \cdot d^{[\sigma]}_{d \oplus e \oplus f^{\sigma+1}} \otimes_1 i\,, \quad \text{with } \sigma = \tau + |\epsilon| \pmod 2\ , \\
s_{YZ} &= (\sigma + 1)\theta(f, e) + \sigma P(f) + P(de) + P(def) + \langle f, i \rangle + \left\langle d, \epsilon \theta(e) \theta(f)^{\sigma+1} \right\rangle\ ; \\
X^+_{d \cdot \delta} &\xrightarrow{g^{-1}} (-1)^{s_T} \cdot X^+_{d \cdot \delta \delta'}\,, \quad \text{for } |d| = 8,\ \delta \text{ even}\,, \quad \text{with } \delta' = A(d, f)\,, \quad \text{and} \\
s_T &= P(e) + P(de) + \langle d, \epsilon \rangle + \langle ef, \delta \rangle + |\delta||\epsilon|/2\ ; \\
X_{\Omega^m \cdot ij} &\xrightarrow{g^{-1}} (-1)^{m|\epsilon| + \langle ef, ij \rangle} \cdot X_{\Omega^n \cdot ij}\ , \quad \text{with } n = m + \langle f, ij \rangle\ ; \\
(ij)_1 &\xrightarrow{g^{-1}} (-1)^{\langle f, ij \rangle} (ij)_1\ .
\end{aligned}$$

The sign bits s_X, s_{YZ}, and s_T can also be computed as follows:

$$\begin{aligned}
s_X &= |\epsilon| \langle d, i \rangle + \langle e, i \rangle + |\epsilon| P(d) + \langle ef, \theta(d) \rangle + \langle d, \epsilon \theta(e) \rangle + P(f)\,; \\
s_{YZ} &= \langle f, i \rangle + \langle f, \theta(d) \rangle + \left\langle d, \epsilon \theta(ef) \theta(f)^{\sigma+1} \right\rangle \\
&\quad + \langle f, \theta(e) \rangle + \sigma \langle e, \theta(f) \rangle + (\sigma + 1) P(f)\,; \\
s_T &= |\delta||\epsilon|/2 + \langle ef, \delta \rangle + \langle e, \theta(d) \rangle + \langle d, \epsilon \theta(e) \rangle\ .
\end{aligned}$$

The operation of the triality elements τ^e

The operation $g = \tau^e$ on ρ_p is coded in function `mm_op<p>_t` in the automatically generated C file `mm<p>_op_t.c` for the modulus `p`. Since τ has order 3, it suffices to present the formulas for $e = \pm 1$. Note that τ fixes $(ii)_1$. From the discussion in [Sey20], Section 8, we obtain for $i \neq j$:

$$\begin{array}{ccccccc}
(ij)_1 & \xrightarrow{\tau} & X_{ij} - X^+_{ij} & \xrightarrow{\tau} & X_{ij} + X^+_{ij} & \xrightarrow{\tau} & (ij)_1\,, \\
2X_{ij} & \xrightarrow{\tau} & (ij)_1 + X_{ij} + X^+_{ij} & \xrightarrow{\tau} & (ij)_1 + X_{ij} - X^+_{ij} & \xrightarrow{\tau} & 2X_{ij}\,, \\
2X^+_{ij} & \xrightarrow{\tau} & (ij)_1 - X_{ij} - X^+_{ij} & \xrightarrow{\tau} & -(ij)_1 + X_{ij} - X^+_{ij} & \xrightarrow{\tau} & 2X^+_{ij}\,, \\
X^+_{d \cdot i} & \xrightarrow{\tau} & (-1)^{\langle d, i \rangle} (d^- \otimes_1 i) & \xrightarrow{\tau} & (-1)^{P(d)} (d^+ \otimes_1 i) & \xrightarrow{\tau} & X^+_{d \cdot i}\,, \\
X^+_{d,\delta} & \xrightarrow{\tau} & \frac{1}{8} \sum_\epsilon (-)_{\delta,\epsilon} X^+_{d,\delta} & \xrightarrow{\tau} & \frac{1}{8} \sum_\epsilon (-)_{\epsilon,\delta} X^+_{d,\delta} & \xrightarrow{\tau} & X^+_{d,\delta}\,.
\end{array}$$

The last formula describes the operation of τ on $X^+_{d,\delta}$, where d is a preimage of an octad in the Parker loop, and δ is an even element of the Golay cocode represented by a subset of that octad. The summation runs over all even subsets ϵ of that octad; and $(-)_{\delta,\epsilon}$ means $(-1)^{|\delta \cap \epsilon| + |\delta|/2}$ as in [Sey20]. We will briefly comment on the implementation of the last formula.

For any octad d as above we consider the row vector X^+_d with 64 entries $X^+_{d,\delta}$, where δ runs over the 64 even subsets of the octad corresponding to d. Then operation of τ on X^+_d can be written as

$$X^+_d \stackrel{\tau}{\longrightarrow} \tfrac{1}{8} X^+_d \cdot D \cdot H \stackrel{\tau}{\longrightarrow} \tfrac{1}{8} X^+_d \cdot H \cdot D \stackrel{\tau}{\longrightarrow} X^+_d \; .$$

Here matrix D is a diagonal matrix with an entry $(-1)^{|\delta|/2}$ in the row and column corrsponding to δ. Matrix H is a Hadamard matrix with an entry $(-1)^{|\delta \cap \epsilon|}$ in the row and column corresponding to δ and ϵ, respectively.

When implementing the muliplication with matrices D and H we have to be aware of the numbering of the entries δ used in Section *The representation of the Monster group*. Using that numbering, it turns out that matrix D has diagonal entries $(-1)^{w(i(i+1)/2)}$, with i running from 0 to 63, and $w(j)$ the bit weight of the binary representation of a nonnegative integer j. Matrix H is a (commuting) product of a permutation matrix P, and a standard 64×64 Hadamard matrix H_{64} according to Sylvester's construction. Matrix P fixes entry i if $w(i)$ is even and exchanges entry i with entry $63 - i$ otherwise.

Here multiplication with matrix H_{64} is the most time-consuming operation. We can speed this up by using the decomposition:

$$H_{64} = H_2 \otimes H_2 \otimes H_2 \otimes H_2 \otimes H_2 \otimes H_2 \, , \quad \text{with } H_2 = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} .$$

Monomial operation of the generators ξ^e

Generator ξ is an element of the subgroup G_{x0} of the monster. The basis vectors with tags `B, C, T, X` (as explained below) of the representation ρ of the Monster span a subspace 98280_x of ρ. The basis vectors with the tags listed above (together with their opposite vectors) correspond to the short elements of the normal subgroup Q_{x0} of G_{x0} of structure 2^{1+24}. The group G_{x0} operates on these basis vectors in the same way as it operations on Q_{x0} by conjugation. Thus G_{x0} (and hence also $\xi^{\pm 1}$) operates monomially on the basis vectors of 98280_x.

We have $Q_{x0}/\{\pm 1\} \cong \Lambda/2\Lambda$. An element x of Q_{x0} is short if it corresponds to a vector in the Leech lattice Λ of norm 4. The operation of $\xi^{\pm 1}$ on Q_{x0} (and hence also of 98280_x) is given by Lemma 9.5 in [Sey20].

In section *The representation of the Monster group* we use the following names and tags for the basis vectors of 98280_x.

Name	Tag	Entries	Remarks
X^+_{ij}	B	i, j; 0 <= j < i < 24	(1)
X_{ij}	C	i, j; 0 <= j < i < 24	(1)
$X_{o,s}$	T	o, s; 0 <= o < 759, 0 <= s < 64	(2)
$X_{d,j}$	X	d, j; 0 <= d < 2**11, 0 <= j < 24	(1),(3)

Remarks

(1) i and j, 0 <= i,j < 24 refer to basis vectors of the Boolean vector space in which the Golay code is defined.

(2) o is one of 759 octads, s is one of 64 even subsets of octad d (modulo its complement in d), as described in section *Octads and suboctads*.

(3) d with 0 <= d < 2048 refers to the Parker loop element d.

We group these basis vectrors into five boxes (labelled 1, …, 5) with each box containing at most 24576 entries. Element ξ permutes these boxes as follows:

$$\mathrm{Box1} \rightarrow \mathrm{Box1}\,, \;\mathrm{Box2} \rightarrow \mathrm{Box2}\,, \;\mathrm{Box3} \rightarrow \mathrm{Box4} \rightarrow \mathrm{Box5} \rightarrow \mathrm{Box3}\,.$$

The mapping from the basis vectors to entries in boxes is:

Basis vector	Box	Entry
B[i,j]	1	0 + 32 * i + j
C[i,j]	1	768 + 32 * i + j
T[o,s], o < 15	1	1536 + 64 * o + s
T[o,s], 15 <= o < 375	2	64 * (o - 15) + s
T[o,s], o >= 375	3	64 * (o - 375) + s
X[d,j], d < 1024	4	32 * d + j
X[d,j], d >= 1024	5	32 * (d - 1024) + j

This subdivision looks weird, but is has quite a few advantages:

- The lower index (j, or s) has stride 1, and the stride of the higher index (i, o, or d) is the lowest possible power of two. So accessing an entry is easy. In the C code for a representation derived from ρ the entries of a vector of such a representation are strided in the same way. So the tables computed under these assumptions can be used in the C code for the operator ξ.
- Boxes are permuted as above, so the operation of $\xi^{\pm 1}$ can be stored in a set ot 5 tables encoding mappings between boxes.

A similar, but finer subdivision of the whole space ρ (and not only of the subspace 98280_x) is given in [Iva09], section 3.4.

We further subdivide a box into *clusters*, *rows*, and *columns*. Here a row always contains 32 columns; with either all or just the first 24 columns being used. A cluster contains a variable nubmer of rows, and a box contains a variable numer of clusters, as indicated in the following table. We also assign names to the boxes.

Box	Name	Size	Clusters	Rows	Columns used
1	BC	2486	1	78	32
2	T0	23040	45	16	32
3	T1	24576	64	12	32
4	X0	24576	64	16	24
5	X1	24576	64	16	24

If a box has c clusters, m rows, and n used columns, then the triple (c, m, n) is called the *shape* of the box.

It turns out that ξ always maps the i-th cluster of a box to the i-th cluster of a (possibly different) box. This greatly simplifies the implementation of the mapping. So it suffices to specify the mapping as a collection of permutations inside a cluster. Since a cluster has size at most 2486, such a permutation can be stored in an array of 16-bit integers, with indices relative to the beginning of the cluster.

An optimized implementation of a mapping $\xi^{\pm 1}$ between two boxes depends on the shape of these two boxes. We want to use the same program code for both mappings, using different parameters in both cases. Therefore we implement five procedures, labelled with a *stage* counting form 0 to 4. These procedures map the boxes as follows:

Stage	ξ :	ξ^{-1} :	Mapping of shapes:
0	BC -> BC	BC -> BC	(1, 78, 32) -> (1, 78, 32)
1	T0 -> T0	T0 -> T0	(45, 16, 32) -> (45, 16, 32)
2	T1 -> X0	T1 -> X1	(64, 12, 32) -> (64, 16, 24)
3	X0 -> X1	X1 -> X0	(64, 16, 24) -> (64, 16, 24)
4	X1 -> T1	X0 -> T1	(64, 16, 24) -> (64, 12, 32)

The functions in module `gen_xi_functions.c` deal with the operation of the generator ξ of the Monster on Q_{x0} and 98280_x. They are also used for computing tables describing the monomial operation of $\xi^{\pm 1}$ on 98280_x. Module `mmgroup.dev.generators.gen_xi_ref` is a pure python substitute for this set of C functions; but calculating the tables with python takes a rather long time.

Class `MM_TablesXi` in module `mmgroup.dev.mm_basics.mm_tables_xi` contains 3-dimensional arrays that can be used for computing the monomial operation of $\xi^{\pm 1}$. In each array the first index (running from 0 to 4) indicates the stage of the computation. as shown in the previuos table.

The following table decribes the entries of the arrays PERM_TABLES, SIGN_TABLES, OFFSETS, and SHAPES in that class.

Table 1: Arrays encoding the monomial operation $\xi^{\pm 1}$

Array name	Shape	Meaning
PERM_TABLES	(5, 2, c*m*n)	Entry (s, e, h*m*n + i*n + j) is the index of the preimage of output entry (h,i,j) in stage s under ξ^{e+1}; with [c, m, n] = SHAPES[s, 1].
SIGN_TABLES	(5, 2, c*m)	Bit j of entry (s, e, h*m + i) is the sign of the preimage of output entry (h,i,j) in stage s under ξ^{e+1}; with [c, m, n] = SHAPES[s, 1].
OFFSETS	(5, 2, 2)	Entries (s, e, 0) and (s, e, 1) are the offests of the source box and the destination box when computing ξ^{e+1} in stage s.
SHAPES	(5, 2, 3)	Entries (s, 0) and (s, 1) are triples encoding the shape of the source and the destiniation box at stage s.

Here we abbreviate the phrase "the entry in cluster h, row i, column j, of the output vector computed in stage s" to "output entry (h,i,j) in stage s". Entries in array PERM_TABLES are indices relative to the beginning of the current cluster.

The following code block multiplies a vector `v_in` in the representation ρ of the Monster with ξ^e for $e = 1, 2$ and stores the monomial part of the result in the vector `v_out`. It uses the arrays described in the table above. The function in that code block is tested in module `mmgroup.tests.test_mm_op.test_table_xi`.

```python
from mmgroup.dev.mm_basics.mm_tables_xi import MM_TablesXi
tbl = MM_TablesXi()
PERM_TABLES = tbl.PERM_TABLES
SIGN_TABLES = tbl.SIGN_TABLES
OFFSETS = tbl.OFFSETS
SHAPES = tbl.SHAPES

def map_xi(v_in, e, v_out):
    r"""Compute v_out = v_in * \xi**e, for e = 1, 2.

    We compute the monomial part of v_out only.
    v_in is a array of integers corresponding to a vector in the
    representation \rho of the Monster. Vector v_out is
```

(continues on next page)

(continued from previous page)

```
    an empty vector of integers of the same size.
    """
    for stage in range(5):
        box_in = v_in[OFFSETS[stage][e-1][0]:]
        box_out = v_out[OFFSETS[stage][e-1][1]:]
        shape_in = SHAPES[stage][0]
        shape_out = SHAPES[stage][1]
        cluster_in_size = shape_in[1] * 32
        cluster_out_size = shape_out[1] * 32
        cluster_perm_size = shape_out[1] * shape_out[2]
        cluster_sign_size = shape_out[1]
        perm_table = PERM_TABLES[stage][e-1]
        sign_table = SIGN_TABLES[stage][e-1]
        for cluster in range(shape_out[0]):
            cluster_in = box_in[cluster * cluster_in_size:]
            cluster_out = box_out[cluster * cluster_out_size:]
            cluster_perm = perm_table[cluster * cluster_perm_size:]
            cluster_sign = sign_table[cluster * cluster_sign_size:]
            for i in range(shape_out[1]):
                for j in range(shape_out[2]):
                    x = cluster_in[cluster_perm[shape_out[2]*i + j]]
                    x = (-1)**(cluster_sign[i] >> j) * x
                    cluster_out[32 * i + j] = x
```

Non-monomial operation of the generators ξ^e

This subsection is yet to be written!

Conjugation of $\tilde{x}_d x_\delta$ with y_e

Sometimes we have to conjugate an element $\tilde{x}_d x_\delta$, where $\tilde{x}_d = x_d x_{\theta(d)}$, with y_e. We have

$$\begin{aligned} y_e^{-1} \tilde{x}_d x_\delta y_e &= x_{-1}^{\alpha} x_\Omega^{\beta} \tilde{x}_d \tilde{x}_e^{|\delta|} x_\delta x_\epsilon \,, \\ \alpha &= \theta(d,e) + \langle e, \delta \rangle^{1+|\delta|} + \mathrm{sign}(e) \,, \\ \beta &= \theta(e,d) + \langle e, \delta \rangle + P(e)^{|\delta|} \,, \\ \epsilon &= A(d,e) + \theta(e)^{|\delta|} \,. \end{aligned}$$

2.3.3 Computations in the Leech lattice modulo 2

In this section we describe the operation of the Conway group Co_1 on $\Lambda/2\Lambda$, where Λ is the Leech lattice. The description of these orbits is taken from [Sey22], Appendix A.

This operation is important for computing in the subgroup G_{x0} of structure $2^{1+24}.\mathrm{Co}_1$ of the monster, as described in [Con85] and [Sey20]. Let Q_{x0} be the normal subgroup of G_{x0} of structure 2^{1+24}. The group $\mathrm{Co}_1 = G_{x0}/Q_{x0}$ is the automorphism group of $\Lambda/2\Lambda$. We assume that Co_1 and also G_{x0} operate on $\Lambda/2\Lambda$ by right multiplication. Let Ω in $\Lambda/2\Lambda$ be the type-4 vector corresponding to the the standard coordinate frame in Λ. The stabilizer of Ω is a subgroup N_{x0} of G_{x0} of structure $2^{1+24}.2^{11}.M_{24}$. Thus the set of type-4 vectors in $\Lambda/2\Lambda$ corresponds the set of right cosets $2^{11}.M_{24}\backslash\mathrm{Co}_1$. We have $N_{x0}\backslash G_{x0} \cong 2^{11}.M_{24}\backslash\mathrm{Co}_1$. So identifying the right coset $N_{x0}h$ for a $h \in G_{x0}$ reduces to the computation of the frame Ωh.

In our construction of the monster the element h may be given as an arbitrary word in the generators of G_{x0} (modulo Q_{x0}). So it is important to find a short word g in the the generators of G_{x0} with $g \in N_{x0}h$. This can be achieved by applying a sequences $g' = g'_1, \ldots, g'_k$ of automorphisms of $\Lambda/2\Lambda$ to Ωh such that $\Omega h g' = \Omega$, and each g'_i corresponds to a generator of G_{x0} (modulo Q_{x0}).

We assume that the reader is familiar with the Conway group Co_1 and the geometry of the Leech lattice as described in [Iva99], section 4.1 - 4.7.

Orbits of the group N_{x0} in the Leech lattice mod 2

In this subsection we describe the orbits of N_{x0} on $\Lambda/2\Lambda$. The type of a vector in $v \in \Lambda$ is the halved scalar product $\frac{1}{2}\langle v, v\rangle$. The type of a vector in $\Lambda/2\Lambda$ is the type of its shortest representative in Λ. Each vector in $\Lambda/2\Lambda$ has type 0, 2, 3 or 4; and the group Co_1 is transitive on the vectors of any of these types.

The orbits of the groups N_{x0} on the vectors of type 2,3, and 4 on Λ have been described in [Iva99], Lemma 4.4.1. N_{x0} acts monomially on the the lattice $\sqrt{8}\Lambda$, which has integers coordinates in the standard Euclidean basis. Thus an orbit of N_{x0} on $\Lambda/2\Lambda$ can be described by the *shapes* of the shortest vectors in the corresponding orbit of $\sqrt{8}\Lambda$. Here the shape of a vector is the multiset of the absolute values of the coordinates of the vector. E.g. a vector of shape $(3^5 1^{19})$ has 5 coordinates with absolute value 3 and 19 coordinates with absolute value 1.

A vector of type 2 or 3 in $\Lambda/2\Lambda$ has two opposite representatives of the same type in Λ; so its shape is uniquely defined. A vector of type 4 in $\Lambda/2\Lambda$ has $2 \cdot 24$ representatives of type 4 in Λ which are orthogonal except when equal or opposite. It is well known that a type-4 vector in $\Lambda/2\Lambda$ corresponds to a coordinate frame in the Leech lattice in standard coordinates, see e.g. [CS99], [Iva99].

The table at Lemma 4.4.1. in [Iva99] assigns a name and a shape to each orbit of N_{x0} on the vectors of type 2, 3, and 4 in Λ. The table at Lemma 4.6.1. in [Iva99] assigns a name and one or more shapes to each orbit of N_{x0} on the vectors of type 4 in $\Lambda/2\Lambda$. We reproduce this information for the orbits of N_{x0} on $\Lambda/2\Lambda$ in the following table. Here we also assign a subtype (which is a 2-digit number) to each orbit. The first digit of the subtype specifies the type of the orbit and the second digit is used to distinguish between orbits of the same type.

Let $\mathcal{C}$ be the Golay code and $\mathcal{C}^*$ is the Golay cocode. For $d \in \mathcal{C}, \delta \in \mathcal{C}^*$ let $\lambda_d, \lambda_\delta \in \Lambda/2\Lambda$ be defined as in [Sey20], Theorem 6.1. Conway's definition of $\lambda_d, \lambda_\delta$ in [Con85] differs slightly from that definition. Each $v \in \Lambda/2\Lambda$ has a unique decomposition $v = \lambda_d + \lambda_\delta, d \in \mathcal{C}, \delta \in \mathcal{C}^*$. The subtype of a vector $\lambda_d + \lambda_\delta \in \Lambda/2\Lambda$ can be computed from d and δ, as indicated in the following table:

Subtype	Name	Shape	$\lvert d\rvert$	$\lvert\delta\rvert$	$\langle d, \delta\rangle$	Remark
00		(0^{24})	0	0	0	
20	Λ_2^4	$(4^2 0^{22})$	$0, 24$	2	0	
21	Λ_2^3	$(3\,1^{23})$	any	1	$\lvert d\rvert/4$	
22	Λ_2^2	$(2^8 0^{16})$	$8, 16$	even	0	1.
31	Λ_3^5	$(5\,1^{23})$	any	1	$\lvert d\rvert/4+1$	
33	Λ_3^3	$(3^3 1^{21})$	any	3	$\lvert d\rvert/4$	
34	Λ_3^4	$(4\,2^8 0^{15})$	$8, 16$	even	1	
36	Λ_3^2	$(2^{12} 0^{12})$	12	even	0	
40	$\bar\Lambda_4^{4a}$	$(4^4 0^{20})$	$0, 24$	4	0	
42	$\bar\Lambda_4^6$	$(6\,2^7 0^{16}), (2^{16} 0^8)$	$8, 16$	even	0	2.
43	$\bar\Lambda_4^5$	$(5\,3^2 1^{21}), (3^5 1^{19})$	any	3	$\lvert d\rvert/4+1$	
44	$\bar\Lambda_4^{4b}$	$(4^2 2^8 0^{14}), (2^{16} 0^8)$	$8, 16$	even	0	3.
46	$\bar\Lambda_4^{4c}$	$(4\,2^{12} 0^{11})$	12	even	1	
48	$\bar\Lambda_4^8$	$(8\,0^{23})$	24	0	0	

Remarks

1. $\lvert\delta\rvert/2 = 1 + \lvert d\rvert/8 \pmod 2$, $\delta \subset d\Omega^{1+\lvert d\rvert/8}$ for a suitable representative δ of the cocode element.
2. $\lvert\delta\rvert/2 = \lvert d\rvert/8 \pmod 2$, $\delta \subset d\Omega^{1+\lvert d\rvert/8}$ for a suitable representative δ of the cocode element.

3. None of the conditions stated in Remarks 1 and 2 hold.

Here column *Subtype* lists the two-digit number that we use for describing the orbit. Columns *Name* and *Shape* list the names and the shapes of the orbits as given in [Iva99], Lemma 4.1.1 and 4.6.1. Columns $|d|$ and $|\delta|$ list conditions on the weight of a Golay code word d and of (a shortest representative of) the Golay cocode element δ, respectively. Column $\langle d, \delta \rangle$ lists conditions on the scalar product of $|d|$ and $|\delta|$. All this information can easily be derived from [Iva99] and [Con85] (or [Sey20]).

The table provides enough information for effectively computing the subtype of an element $\lambda_d + \lambda_\delta$ from d and δ. Function `gen_leech2_type` in file `gen_leech.c` computes that subtype. It returns e.g. the subtype 46 as the hexadecimal number 0x46.

The following table may be helpful for memorizing the second digit of a subtype of a vector in $\Lambda/2\Lambda$.

Subtype	Description
$x0$	Contains a vector $\lambda_d + \lambda_\delta$ with $d = 0$; δ even
$x1$	Vectors $\lambda_d + \lambda_\delta$ with $|\delta| = 1$
$x2$	Vectors $\lambda_d + \lambda_\delta$ with $|d| = 8, 16$; and $\delta \subset d\Omega^{1+|d|/8}$, $|\delta|$ even
$x3$	Vector $\lambda_d + \lambda_\delta$ with $|\delta| = 3$
$x4$	Vectors $\lambda_d + \lambda_\delta$ with $|d| = 8, 16$; not of subtype $x2$, $|\delta|$ even
$x6$	Vectors $\lambda_d + \lambda_\delta$ with $|d| = 12$; $|\delta|$ even
$x8$	The vector $\lambda_d + \lambda_\delta$ with $|d| = 24$; $\delta = 0$

Operation of the group G_{x0} on the Leech lattice

The generators $x_d, x_\delta, y_d, x_\pi, \xi$, with $d \in \mathcal{C}, \delta \in \mathcal{C}^*, \pi \in \mathrm{Aut}_{St}\mathcal{P}$ operate on the subgroup G_{x0} of the monster and also on $\Lambda/2\Lambda$, as described in [Sey20]. Here x_d, x_δ operate trivially on $\Lambda/2\Lambda$. y_d and x_π act as permutations and sign changes on the coordinates of Λ, respectively; they do not change the subtype.

From the *Leech graph* in [Iva99], section 4.7, we see how the generators ξ and ξ^2 may change the subtype of a type-4 vector in $\Lambda/2\Lambda$. The following figure shows a subgraph of that graph. It shows the transitions of subtypes of type-4 vectors (due to transformations by ξ or ξ^2) that we actually use. We will provide some more information about the

$$\begin{tikzcd}
\boxed{43} \arrow[r] \arrow[ddr] & \boxed{42} \arrow[dr] & & \\
& & \boxed{40} \arrow[r] & \boxed{48} \\
\boxed{46} \arrow[r] & \boxed{44} \arrow[ur] & &
\end{tikzcd}$$

Fig. 1: Transitions of subtypes of type-4 vectors in the Leech lattice mod 2

operation of ξ^k on Λ.

We say that a vector $w \in \mathbb{Z}^n$ is of shape $(m^\alpha 0^{n-\alpha} \bmod 2m)$ if w has α coordinates equal to $m \pmod{2m}$ and $n - \alpha$ coordinates equal to $0 \pmod{2m}$. For a vector $v = (v_0, \ldots, v_{23}) \in \mathbb{R}^{24}$ define $(v_{4i}, v_{4i+1}, v_{4i+2}, v_{4i+3}) \in \mathbb{R}^4$ to be the i-th column of v. This definition is related to the Miracle Octad Generator (MOG) used for the description of the Golay code and the Leech lattice, see [CS99], chapter 11. The following lemma has been shown in [Sey22].

Lemma

Let $v \in \sqrt{8}\Lambda$, and let $v^{(k)} = v \cdot \xi^k$. Let w be a column of v, and let $w^{(k)}$ be the corresponding column of $v^{(k)}$. Then $|w^{(k)}| = |w|$. If w has shape $(m^4 \bmod 2m)$ then there is a unique $k \in \{1, 2\}$ such that $w^{(k)}$ has shape $(0^4 \bmod 2m)$. If w has shape $(m^2 0^2 \bmod 2m)$ then $w^{(k)}$ has shape $(m^2 0^2 \bmod 2m)$ for all $k \in \mathbb{Z}$.

Computing a transversal of $G_{x0} \backslash N_{x0}$

The cosets in $G_{x0} \backslash N_{x0}$ correspond to the type-4 vectors in the Leech lattice modulo 2. Thus for computing a transversal of $G_{x0} \backslash N_{x0}$ we have to compute an element of G_{x0} that maps the standard frame Ω in $\Lambda/2\Lambda$ to a vector v for a given vector v of type 4 in $\Lambda/2\Lambda$.

In the next few subsections we show how to map a vector of subtype 4x to a vector of subtype 48 by a sequence of generators x_π, ξ^k of length at most 6. The proofs are a bit involved, but the C function `gen_leech2_reduce_type4_vector` in file `gen_leech.c` constructs that sequence very fast. Then the inverse of the element computed by function `gen_leech2_reduce_type4_vector` is the requested element of the transversal.

If we select a permutation in the Mathieu group M_{24} then we actually have to select an element of the group $\mathrm{Aut}_{\mathrm{St}} \mathcal{P}$ of standard automorphisms of the Parker loop described in *Automorphisms of the Parker loop*. Whenever we have to select a permutation mapping a certain set to another set, we want to keep the inverse of the selected element of $\mathrm{Aut}_{\mathrm{St}} \mathcal{P}$ as simple as possible. More specifically, for that inverse we select the standard representative in $\mathrm{Aut}_{\mathrm{St}} \mathcal{P}$ of the least feasible permutation in lexicographic order. We use the function `mat24_perm_from_map` in file `mat24_functions.c` for finding a suitable permutation in M_{24}.

Incorporating the baby monster into the computation of a transversal

The capability of computing in a subgroup H^+ of structure $2.B$ of the monster $\mathbb{M}$ (where B is the baby monster) is crucial for the success of our project. Although not strictly necessary, we want to select elements of H^+ in our transversal of $G_{x0} \backslash N_{x0}$ whenever possible. In this subsection we explain how to do so. This is a bit technical and may be skipped at first reading.

Let $\beta = \frac{1}{\sqrt{8}}(e_2 - e_3) \in \Lambda$, where e_i is the i-th unit vector in $\mathbb{R}^{24}$. Then β is of type 2 in the Leech lattice Λ, and the centralizer H^+ of x_β in $\mathbb{M}$ has structure $2.B$, and the centralizer H of x_β in G_{x0} has structure $2^{1+22}.\mathrm{Co}_2$. Our goal is find a representative of a coset in $N_{x0} \backslash G_{x0}$ in H (centralizing the element x_β of $\mathbb{M}$) whenever possible.

The coset in $N_{x0} \backslash G_{x0}$ corresponding to a vector $v \in \Lambda/2\Lambda$ of type 4 has a nonempty intersection with H if and only if $v + \beta$ is of type 2 and orthogonal to β in the real Leech lattice Λ, see [Sey22] for details.

If this is the case then we compute an element of $H \subset G_{x0}$ that maps $v + \beta$ to the type-2 vector $\Omega + \beta$ and fixes β.

Note that the generator ξ fixes β. A permutation $\pi \in M_{24}$ fixes β if and only if it fixes the set $\{2, 3\}$ of points.

From subtype 46 to subtype 44

Let $\lambda_r = \lambda_d + \lambda_\delta$ be of subtype 46. Then d is a dodecad. Assume that d contains one column of the MOG. Then up to signs and permutation of the columns of the MOG, and up to permutation of the entries of a column, there is a vector $v \in \sqrt{8}\Lambda$ with $v/\sqrt{8} = \lambda_r \pmod{2\Lambda}$ that has MOG coordinates

2	0	2	0	0	0
2	0	2	0	0	0
2	0	4	2	2	2
2	0	0	2	2	2

.

By the Lemma given above there is a $k \in \{1, 2\}$ such that column 0 of the the vector $w = v \cdot \xi^k$ has shape $(4\,0^3)$. The other columns of w have the same shape as the corresponding columns of v. Thus w has shape $(4\,2^8 0^{15})$. So using the table given above we see that w has type 44.

Note that both, the dodecad d and its complement, contain exactly one column of the MOG. The union of these two columns is a grey even octad o.

With a little extra effort we can show $k = 2 - \langle o, \delta \rangle$, where $\langle ., . \rangle : \mathcal{C} \times \mathcal{C}^* \to \{0, 1\}$ is the scalar product.

If dodecad d does not contain a column of the MOG then we select a permutation that maps the first four entries of dodecad d to the points $0, 1, 2, 3$, i.e. to the first column of the MOG. Then we proceed as above.

From subtype 43 to subtype 42 or 44

Let $\lambda_r = \lambda_d + \lambda_\delta$ be of subtype 43. Then $|\delta| = 3$. The three points of δ can lie in one, two, or three different columns of the MOG.

If all entries of δ lie in the same column of the MOG then up to signs and permutation of the columns of the MOG, and up to permutation of the entries of a column, there is a vector $v \in \sqrt{8}\Lambda$ with $v/\sqrt{8} = \lambda_r \pmod{2\Lambda}$ that has MOG coordinates

5	1	1	1	1	1
3	1	1	1	1	1
3	1	1	1	1	1
1	1	1	1	1	1

.

By the Lemma given above there is a $k \in \{1, 2\}$ such that one column of $w = v \cdot \xi^k$ (and hence all columns) have even entries. Column 0 of w has squared norm $44 = 6^2 + 2^2 + 2^2 + 0^2$. That decomposition of 44 into a sum of four even squares is unique, so column 0 has shape $(6\,2^2 0)$. The other columns have shape $(2\,0^3)$. So w is of shape $(6\,2^7 0^{15})$ and hence of subtype 42.

With a little extra effort we can show $k = 2 - \langle d, \omega \rangle$, where ω is the standard tetrad represented by $\{0, 1, 2, 3\}$, and $\langle ., . \rangle$ as above.

If the entries of δ lie in two different columns of the MOG then up to signs and permutations as above, the corresponding vector in $v \in \sqrt{8}\Lambda$ has MOG coordinates

3	5	1	1	1	1
3	1	1	1	1	1
1	1	1	1	1	1
1	1	1	1	1	1

.

By a similar argument as in the first case there is a $k \in \{1, 2\}$ such that the first two columns of $w = v \cdot \xi^k$ have shape $(4\,2\,0^2)$ and $(4\,2^3)$, and that the other columns of w have shape $(2\,0^3)$. Thus w is of subtype 44.

If the entries of δ lie in three different columns of the MOG then we apply a permutation in M_{24} that maps δ to $\{1, 2, 3\}$, and proceed as in the first case.

From subtype 44 to subtype 40

Let $\lambda_r = \lambda_d + \lambda_\delta$ be of subtype 44. Then d is an octad or a complement of an octad. We call that octad o. If the cocode word δ is a duad, let c_0, c_1 be the two points in that duad. Otherwise, let c by any tetrad intersecting the octad o in two points, so that c is equal to the sextet δ modulo the cocode $\mathcal{C}^*$. Let c_0, c_1 be the two points in c which are not in the octad o.

Assume first that o is a grey even octad, i.e. a union of two columns of the MOG, and that the points c_0, c_1 are in the same column of the MOG.

Then up to signs and permutation of the columns of the MOG, and up to permutation of the entries of a column, there is a vector $v \in \sqrt{8}\Lambda$ with $v/\sqrt{8} = \lambda_r \pmod{2\Lambda}$ that has MOG coordinates

2	2	4	0	0	0
2	2	4	0	0	0
2	2	0	0	0	0
2	2	0	0	0	0

Similar to the previous cases, we can show that there is a unique $k \in \{1, 2\}$ such that $w = v \cdot \xi^k$ has shape $(4^4 0^{20})$. So w is of subtype 40.

One can show $k = 2 - \sigma$, where $\sigma = |c \cap w| \pmod 2$, $\sigma \in \{0, 1\}$, and w is any of the two columns of the MOG contained in the octad o.

If the octad o and the two points c_0, c_1 do not satisfy the condition given above then we first construct a permutation that maps c_0, c_1 to $2, 3$, and four points of o to $4, 5, 6, 7$ as follows.

We compute the syndrome of the set c_0, c_1, o_0, o_1, o_2. That syndrome intersects with o in exactly one point o_j. The we construct a permutation in M_{24} that maps the hexad $(c_0, c_1, o_0, o_1, o_2, o_j)$ to the hexad $(2, 3, 4, 5, 6, 7)$. Such a permutation exists, since both hexads are subsets of an octad.

From subtype 42 to subtype 40

Let $\lambda_r = \lambda_d + \lambda_\delta$ be of subtype 42. Then d is an octad or a complement of an octad. We call that octad o as in the case of type 42. If that octad is grey, we map vector λ_r to a vector of type 40 in the same way as in that case.

Otherwise we first map the lowest four points of octad o to the set $0, 1, 2, 3$, using a permutation in M_{24}.

From subtype 40 to subtype 48

Let λ_r be of subtype 40. Then $\lambda_r = \alpha\lambda_\Omega + \lambda_\delta$, $\alpha = 0, 1$, for some $\delta \in \mathcal{C}^*$, $|\delta| = 4$. If δ is equal to the standard tetrad ω represented by $\{0, 1, 2, 3\}$ then there is a unique power of ξ that maps λ_r to λ_Ω, which is of type 48.

Otherwise let c be the tetrad containing the point 0 that corresponds to δ. The we first apply a permutation in M_{24} that maps c to $\{0, 1, 2, 3\}$.

From subtype 21 to subtype 22, fixing β

Let $\lambda_r = \lambda_d + \lambda_\delta$ be of subtype 21, and orthogonal to β in the real Leech lattice. Then $|\delta| = 1$.

We proceed as in the case of subtype 43. Then there is a $k \in \{1, 2\}$ such that one column of $w = v \cdot \xi^k$ (and hence all columns) have even entries. It is easy to see that w is of subtype 22.

From subtype 22 to subtype 20, fixing β

Let $\lambda_r = \lambda_d + \lambda_\delta$ be of subtype 22, and orthogonal to β in the real Leech lattice. Then d is an octad or a complement of an octad. We call that octad o as in the cases of subtype 42 or 44. For subtype 22 the cocode word δ can be considered as a subset of octad o.

The set $\{2, 3\}$ either contained in or disjoint to octad o.

If octad o is grey and even then we may directly apply a power of ξ that transforms λ_r to a vector of subtype 22, similar to the operation in the case of subtype 42. Otherwise we proceed as follows.

If the set $\{2, 3\}$ is disjoint from o we proceed exactly as in the case of subtype 44, replacing c_0 and c_1 by 2 and 3. This operation fixes the points 2 and 3.

Otherwise we first fix the points 2 and 3 and map the lowest two points of $o \setminus \{2, 3\}$ to 0 and 1, using a permutation in M_{24}.

From subtype 20 to the vector $\beta + \Omega$, fixing β

Let λ_r be of subtype 20, and orthogonal to β in the real Leech lattice. Then $\lambda_r = \alpha\lambda_\Omega + \lambda_\delta$, $\alpha = 0, 1$, for some $\delta \in \mathcal{C}^*$, $|\delta| = 2$. In case $\delta = \{2, 3\}$ we must have $\alpha = 1$, i.e. $\lambda_r = \beta + \Omega$, and we are done. Otherwise both points in δ are different from 2 and 3.

In case $\delta = \{0, 1\}$ the transformation ξ^k with $k = 2 - \alpha$ maps λ_r to β.

Otherwise we first apply a permutation in M_{24} that maps δ to $\{0, 1\}$, and fixes 2 and 3.

2.3.4 Orbits of the group N_{x0} in the group Q_{x0}

Function `gen_leech2_reduce_n` in file `gen_leech_reduce_n.c` computes a standard representative of the orbit of a nonzero vector in the extraspecial subgroup Q_{x0} of G_{x0} under the action of the group N_{x0}. Here N_{x0} acts on the group Q_{x0} of structure 2_+^{1+24} by conjugation.

The quotient of Q_{x0} by its center $\{x_{\pm 1}\}$ is isomophic to $\Lambda/2\Lambda$, i.e. to the Leech lattice mod 2. So we may define the subtype of any $v \in Q_{x0}$ as the subtype of the element $vx_{\pm 1}$ of $\Lambda/2\Lambda$.

With one exception the orbit of a nonzero vector $v \in Q_{x0}$ under N_{x0} depends on the subtype of v only. There are exactly two elements $x_{\pm 1}$ of Q_{x0} of subtype 00, which are of course in different N_{x0} orbits.

For each nonzero orbit of N_{x0} we choose a representative $x_d x_\delta$, $d \in \mathcal{P}, \delta \in \mathcal{C}^*$ (depending on the subtype of the orbit) according to the following table:

Subtype	x_d	δ
00	x_{-1}	$\{\}$
20	x_1	$\{2, 3\}$
21	x_1	$\{0\}$
22	x_o	$\{\}$
31	x_Ω	$\{0\}$
33	x_Ω	$\{1, 2, 3\}$
34	x_o	$\{0, 8\}$
36	x_D	$\{\}$
40	x_1	$\{0, 1, 2, 3\}$
42	$x_\Omega x_o$	$\{\}$
43	x_1	$\{1, 2, 3\}$
44	x_o	$\{8, 9\}$
46	x_D	$\{0, 12\}$
48	x_Ω	$\{\}$

x_o is the (positive) element of the Parker loop corresponding to the stadard octad $\{0, 1, 2, 3, 4, 5, 6, 7\}$.

x_D is the (positive) element of the Parker loop corresponding to the stadard dodecad $\{0, 4, 8, 13, 14, 15, 17, 18, 19, 21, 22, 23\}$.

This table can be computed by function `gen_leech2_reduce_n_rep` in file `gen_leechc_reduce_n.c`.

2.3.5 Checking equality of monster elements and membership in 2^{1+24}.Co$_1$

Checking equality of monster elements

For checking equality of two elements of the monster we may use two nonzero vectors w_{71} and w_{94} in the 198883-dimensional representation of the monster that are fixed by an element of order 71 and negated by an element of order 94, respectively. In [LPWW98] it is shown that only the neutral element of the monster fixes both vectors, w_{71} and w_{94}. So we may check if an element is the identity in the monster. Here the corresponding calculations can be done modulo any odd prime; and we may even use different primes for the operation on w_{71} and on w_{94}.

The python script `mmgroup.structures.find_order_vectors.py` generates suitable vectors w_{71} and w_{94} at random and stores data for the fast recomputation of these vectors in file `order_vector_data.py`. Since we actually work in the representation ρ of dimension 1 + 196883, we have to make sure that the generated vectors are not fixed by the monster.

More precisely, we generate w_{71} as a vector in the representation ρ_3 of the monster in characteristic 3, and w_{94} as a vector in the representation ρ_5 of the monster in characteristic 5. We combine these two vectors to a vector in the representation ρ_{15} (modulo 15) of the monster via Chinese remaindering. Note that a computation in ρ_{15} is faster than the combination of the corresponding computations in ρ_3. We write w for the vector combined from w_{71} and w_{94}.

Testing membership in 2^{1+24}.Co$_1$

For testing membership in 2^{1+24}.Co$_1$ we impose additional conditions on the vector w_{71} in the representation ρ_3 defined in the last section.

For an odd number p let $\rho_{A,p}$ be the subspace of ρ_p spanned by the basis vectors of ρ_p with tag `A`. Then $\rho_{A,p}$ is fixed by the subgroup G_{x0} of structure 2^{1+24}.Co$_1$ of the monster. A vector in $\rho_{A,p}$ has a natural interpretation as a symmetric bilinear form on the Leech lattice (modulo p), see [Con85]. Let Q_p be the symmetric bilinear form on the Leech lattice (modulo p) given by the scalar product; then Q_p is given by the the 24×24 unit matrix, up to a scalar factor.

Let w be a vector in ρ_{15} as in the last section. For any w in ρ_p, let $w_{(A,p)}$ be the bilinear from on the Leech lattice (modulo p) corresponding to the projection of w onto $\rho_{A,p}$. We shall search for a vector w satisfying the condition stated an the last section. In addition we require that for some integer d (modulo 3) the bilinear form $w_{(A,3)} - d \cdot Q_3$ has a kernel of dimension one, and that this kernel contains a type-4 vector x_4 of the Leech lattice modulo 3.The chance of finding such a vector w is discussed in [Sey22], Appendix B. In average, finding a suitable vector w take less than a minute on the author's compute, when running with 12 processes in parallel.

Let Ω be the standard frame of the Leech lattice (modulo 3). Then Ω is of type 4 and consists of the vectors $(0, \ldots, 0, \pm 1, 0, \ldots, 0)$. Once we have found a suitable vector w, we apply a transformation in G_{x0} to w that maps the frame in the Leech lattice given by x_4 to Ω. Therefore we work in the Leech lattice modulo 2, as described in section *Computations in the Leech lattice modulo 2*. Conversion from a type-4 vector in the Leech lattice mod 3 to a type-4 vector in the Leech lattice mod 2 is easy; the C function `gen_leech3to2_type4` in file `gen_leech3.c` does that job. In the sequel we also write w for the transformed vector. So we may assume that the kernel of $w_{(A,3)} - d \cdot Q_3$ is one dimensional and spanned by a vector in the standard frame Ω of the Leech lattice modulo 3.

For doing linear algebra in $\mathbb{F}_3$ we store a vector of 16 integers modulo 3 in a 64-bit integer; and we use arithmetic, Boolean and shift operations.

Assume that g is in G_{x0}. Then the kernel of $w_{A,3} \cdot g - d \cdot Q_3$ is spanned by a vector in the frame $\Omega \cdot g$. So we may also compute $\Omega \cdot g$ in the Leech lattice modulo 2. Using the methods in section *Computations in the Leech lattice modulo 2* we can compute a h_1 in G_{x0} that maps $\Omega \cdot g$ to Ω. So gh_1 fixes Ω. Hence gh_1 is in the subgroup N_{x0} of structure $2^{1+24+11}.M_{24}$ of G_{x0}.

The factor group M_{24} of N_{x0} acts on the 24×24 matrix representing the bilinear form $w_{(A,p)}$ by permuting the rows and the columns, up to changes of signs that we will ignore at the moment. So we store the multiset of the absolute values of the entries of each row of the matrix $w_{(A,p)}$. Then M_{24} permutes these 24 multisets by its natural action. So we may recover the element of M_{24} performing that action and compute a preimage h_2 of the inverse of that element in

N_{x0}. Thus gh_1h_2 is in the subgroup $2^{1+24+11}$ of N_{x0}. Note that, originally, the vector w has been defined modulo 15, so that almost certainly we have enough information in the matrix $w_{(A,15)}$ to compute the permutation in M_{24}. In the implementation we compute a hash function on the 24 multisets obtained from the rows of $w_{(A,15)}$, and we generate a new vector w if these 24 hash values are not mutually different.

Now that task of reducing an element of $2^{1+24+11}$ is easy. We first eliminate the factor 2^{11} be checking signs in $w_{(A,15)}$, or signs of suitable entries of vector w with tag `A`, which is the same thing. Then we eliminate the factor 2^{24} be checking signs of suitable entries of w with tags `B, C, T, X`, and, finally, the factor 2^1 by checking a sign of an entry with tag `Y` or `Z`.

So we obtain a decomposition $gh_1h_2h_3 = 1$, with h_i in a transversal of G_i/G_{i+1} for $G_1 = G_{x0}$, $G_2 = N_{x0}$, $G_3 = 2^{1+24+11}$, $G_4 = \{1\}$. If $g \notin G_{x0}$ then this procedure fails in one of the steps mentioned above. We abort as early as possible in case of failure, in order to accelerate functions searching for an element of G_{x0} that have a low probability of success.

Module `mmgroup.dev.mm_reduce.find_order_vector.py` contains a function for finding a suitable vector w. Once a vector w has been found, it creates a module `mmgroup.dev.mm_reduce.order_vector_data.py` containing information for recomputing the same vector w more quickly. The code generation process creates a C file `mm_order_vector.c` containing the vector w in its source code together with some data required for using w and for checking its correctness. We believe that this method simplifies the integration of our project into computer algebra systems like GAP or Magma, since these systems may now simply call C functions for computing in the monster group, without bothering how to generate a suitable vector w.

2.3.6 Subgroups of the Mathieu group M_{24}

An amalgam of many (mostly 2-local) large subgroups of the Monster $\mathbb{M}$ has been constructed in [Iva09]. For each of these subgroups (and, hopefully, also for their intersections) we want an algorithm that constructs an (almost) uniform random element of that group. This is a complicated task, which might not even be possible for all cases. As a first step towards this goal we will describe the intersections of some subgroups of the Mathieu group M_{24}. These subgroups of M_{24} are involved in the subgroups of $\mathbb{M}$ in the amalgam. We will also give some information how to construct ramdom elements of these subgroups of M_{24} and of their intersections.

TODO: This section is yet a fragment and will be completed in a future version.

Some subgroups of the the Mathieu group M_{24}

We define some subgroups of the Mathieu group M_{24} as the stabilizers of certain subsets (or sets of subsets) of the set $\tilde{\Omega} = \{0, \ldots, 23\}$, as shown in the following table:

Name	Mnemonic	Stabilizes the set	Structure
M_2	2-set	$\{2, 3\}$	$M_{22} : 2$
M_o	octad	$\{0, \ldots, 7\}$	$2^4 : A_8$
M_t	trio	$\{\{8i, \ldots, 8i + 7\} \mid i < 3\}$	$2^6 : (S_3 \times L_3(2))$
M_s	sextet	$\{\{4i, \ldots, 4i + 3\} \mid i < 6\}$	$2^6 : 3.S_6$
M_l	(line)	$\{\{2i, 2i + 1\} \mid 4 \leq i < 12\}$	$2^{1+6} : L_3(2)$
M_3	3-set	$\{1, 2, 3\}$	$L_3(4) : S_3$

All these groups, except for M_l, are maximal subgrous of M_{24} and discussed in [CS99], Ch. 11. The structure of M_l (as a subgoup of M_o) is described in [Iva09] in the text after Lemma 4.1.3. The central element of M_l of order 2 is obtained by exchanging all entries $2i$ with $2i + 1$ for $i \geq 4$. We have the following inclusions:

$$\begin{array}{lll} M_l \subset M_o\,, & M_l \cap M_2 \subset M_t\,, & M_t \cap M_2 \subset M_o \cap M_l\,, \\ M_3 \cap M_l \subset M_s\,, & M_3 \cap M_t \subset M_o \cap M_s\,, & M_3 \cap M_t \cap M_l \subset M_2\,. \end{array}$$

Membership of an element of M_{24} in any of these subgroups can easily be checked computationally. Thus an inclusion of two intersections of these subgroups can be disproved computationally by finding an element of M_{24} contadicting that inclusion. This computation is done in function `test_mat24_rand` in module `mmgroup.tests.test_mat24` during the standard test.

In the remainder of this subsection we will prove the inclusions given above. The first inclusion is obvious. Next we justify the following two inclusions.

By [CS99], Ch. 11, the group M_o acts on the octad $o = \{0, \ldots, 7\}$ as the alternating group A_8, and on the complement $\bar{o}$ of o as the affine group on $\mathbb{F}_2^4$. Here the affine structure of $\bar{o}$ is given by the last 4 digits of the binary representations of its entries. Fixing e.g. the element 8 of $\bar{o}$, the set $\bar{o}$ acquires the structure of the linear space $\mathbb{F}_2^4$; and we obtain the well-known isomorphism $L_4(2) \cong A_8$. Subspaces of dimension 1, 2, and 3 of the linear space $\mathbb{F}_2^4$ will be called lines, planes, and hyperplanes.

We have following correspondences between o and its complement $\bar{o}$

Linear space $\bar{o} = \mathbb{F}_2^4$	Octad o	Remarks
line	Steiner system $S(3,4,8)$	defines an affine structure on o
plane	sextet refining o	tetrads of the sextet in $\bar{o}$ are planes parallel to that plane
hyperplane	Steiner system $S(3,4,8)$	defines an affine structure on o
symplectic form	set of 2 elements	
line incident with hyperplane	partition into 4 sets of 2 elements	

These correspondences (except for the last) are stated in [CCN+85]. For the last correspondence we refer to [CS99], Ch. 10.1.4. The tetrads of a Steiner system corresponding to a line or a hyperplane are the tetrads in the sextets corresponding to the planes incident with that line or hyperplane; here we take tetrads that are subsets of octad o only. Methods for computing with these objects are presented in [CS99], Ch. 11, and implemented in file `mat24_functions.c`.

Let f be the symplectic form on $\mathbb{F}_2^4$ corresponding to a pair y of elements of o. For two different nonzero elements x_1, x_2 of $\mathbb{F}_2^4$ let s be the sextet corresponding to the plane generated by x_1 and x_2, and let τ be a tetrad in s which is a subset of o . Then the scalar product x_1 and x_2 with respect to f is the size of the set $y \cap \tau$ (modulo 2).

Let $l' = \{8, 9\}, s' = \{8, 9, 10, 11\}$ and $t' = \{8, \ldots, 15\}$. Then the line l', the plane s', and the hyperplane t' are mutually incident. A staightforward calculation using the facts stated above shows that t' is the orthogonal complement of l' with respect to the symplectic form f' corresponding to the subset $\{2, 3\}$ of o. Obviously, $M_t \cap M_2 subset M_o$. So this proves $M_l \cap M_2 \subset M_t$ and $M_t \cap M_2 \subset M_o \cap M_l$.

Next we will show the last three inclusions listed above. The group M_3 fixes the set $y = \{1, 2, 3\}$. We have $M_3 \cap M_l \subset M_o$, so $M_3 \cap M_l$ fixes a Steiner system in o, an it turns out the the set $\{0, 1, 2, 3\} \supset y$ is in that Steiner system. This proves $M_3 \cap M_l \subset M_s$. We obviously have $M_3 \cap M_t \subset M_o$, and hence $M_3 \cap M_t$ fixes another Steiner system in o. Here the set $\{0, 1, 2, 3\}$ is also in this Steiner system, implying $M_3 \cap M_t \subset M_o \cap M_s$.

The group $M_t \cap M_l$ fixes a line incident with a hyperplane in $\mathbb{F}_2^4$. Hence it fixes a partition of o into four unordered pairs, and it turns out that the set u of these pairs is $\{\{2i, 2i+1\} \mid 0 \leq i < 4\}$. Thus $M_3 \cap M_t \cap M_l$ fixes u and y, and hence also the set $\{2, 3\}$. This proves $M_3 \cap M_t \cap M_l \subset M_2$.

Some 2-local subgroups of the Monster

The subgroups of M_{24} constructed above are involved in some large subgrups of the Monster as described in the following table.

Name	Involved in subgroup of $\mathbb{M}$
M_2	$H^+ = 2.\mathrm{B}$
M_o	$G_{10} = 2^{10+16}.O_{10}^+(2)$
M_t	$G_5^{(t)} = 2^{5+10+20}.(S_3 \times L_5(2))$
M_s	$G_3 = 2^{3+6+12+18}.(L_3(2) \times L_3(2))$
M_3	$2^2.{}^2E_6(2) : S_3$

TODO: More details will be given in a future version.

2.4 Installation from a source distribution

The current version of the `mmgroup` package is a source distribution that has been tested on a bit Windows, Linux and macOS with a 64-bit x86 CPU. It runs with python 3.8 or higher. The sources of the project can be downloaded from

https://github.com/Martin-Seysen/mmgroup .

We use the python package `cibuildwheel` to build several wheels on these operating systems. The GitHub repository of the project contains actions to build the corrsponding python wheels. The program code for the GitHub actions is stored in subdirectory `.github/workflows`. More details about the build process are given in section *The build process*.

The *mmmgroup* package contains a number of extensions written in `C` which have to be built before use. This will be discussed in section *Code generation*.

2.4.1 Building the `mmgroup` package with cibuildwheel

The easiest way to build the `mmgroup` package is to clone the sources from the github repository with *git*, and to build a python wheel with *cibuildwheel*. Therefore, the cibuildwheel tool must support your target platform. In the sequel we assume that python and the git revision control system are installed on your platform. We describe the build process on a unix-like platform.

Depending on your platform, you might have to install additional tools. For details we refer to the cibuildwheel documentation https://cibuildwheel.readthedocs.io/en/stable/

Any previously installed version of the `mmgroup` package must be uninstalled before building a new version of that package!

The current build system supports native compilation only. Cross compilation is not supported.

E.g. for building the `mmgroup` package for python 3.12 on Linux with a 64-bit x86 CPU, open a shell and type:

```
pip3 install cibuildwheel
git clone https://github.com/Martin-Seysen/mmgroup
cd mmgroup
git pull origin master
git checkout .
python3 -m cibuildwheel --output-dir wheelhouse --only cp312-manylinux_x86_64
```

The last command builds the python wheel. For building the wheel on a different platform, the argument 'cp312-manylinux_x86_64' in the last command should be replaced by a *build identifier* that describes the requested python version and target platform. A list of valid build identifiers is given in Section *'Options/build selection'* of the cibuildwheel documentation, see

https://cibuildwheel.readthedocs.io/en/stable/options/#build-skip

After building the python wheel, switch to subdirectory *wheelhouse*, and check the file name of the wheel that has been built:

```
cd wheelhouse
ls
```

Then you may install the wheel with pip, e.g.:

```
pip3 install mmgroup-0.0.13-cp312-cp312-manylinux_2_17_x86_64.manylinux2014_x86_64.whl
```

Here the file name after the *pip3 install* command must be changed to the file name of the wheel just generated.

You should test your installation of the `mmgroup` package before using it:

```
pip3 install pytest
python3 -m pytest --pyargs mmgroup -m "not slow"
```

2.4.2 Dependencies

Before you can use this source distribution or build its extensions you should install the following python packages:

Table 2: External Python packages required

Package	Purpose
`cython`	Development: integrating `C` programs into the `mmgroup` package
`numpy`	Runtime: Most arrays used by the `mmgroup` package are `numpy` arrays
`wheel`	Distribution: package for generating a Python wheel
`pytest`	Testing: basic package used for testing
`regex`	Development: parsing balanced curly braces
`setuptools`	Development: basic package used for setup and building extensions
`cibuildwheel`	Development: build wheel in continuous integration process
`auditwheel`	Development (Linux and macOS only): patching shared libraries
`sphinx`	Documentation: basic package used for documentation
`sphinx-rtd-theme`	Documentation: 'theme' to be used by sphinx
`sphinxcontrib-bibtex`	Documentation: bibliography in BibTeX style
`sphinxcontrib-tikz`	Documentation: link between `doxygen` and `sphinx`
`breathe`	Documentation: link between `TikZ` and `sphinx`

Packages used for the purpose of documentation are required only if you want to rebuild the documentation. If you want to rebuild the documentation you should also install the following programs:

Table 3: External programs required

Program	Purpose	Location
miktex	Documentation	https://miktex.org/
Perl	Documentation	https://www.perl.org/get.html
doxygen	Documentation of C files	https://www.doxygen.nl/download.html
Ghostcript	Documentation: using TikZ in Sphinx	https://ghostscript.com/releases/gsdnld.html

2.4.3 Installing the package

To build the required package on your local computer, go to the root directory of the distribution. This is the directory containing the files `setup.py` and `README.rst`. From there run the following commands in a shell:

```
python -m pip install -r requirements.txt
python setup.py bdist_wheel
```

In Linux or macOS you'll have to change the `python` command to `python3`.

For testing the installation, run the following command:

```
python -m pytest ./src/mmgroup/ -Wignore -v -s -m "not slow"
```

Distributing a *wheel* is the standard way to distribute a python package, see e.g.

https://packaging.python.org/guides/distributing-packages-using-setuptools/#wheels

Remarks

If you have installed any version of the `mmgroup` package (e.g. with the `pip` tool) then you **must** uninstall that package before you can build a new version of the `mmgroup` package from the source files.

2.4.4 Installing a C compiler for cython in Windows

The bad news for Windows developers is that there is no pre-installed `C` compiler on a standard Windows system. However, the `cython` package requires the `C` compiler `MSVC`.

The user has to install a `C` compiler so that it cooperates with `cython`. That installation process is out of the scope of this document.

For installing `MSVC`, one might start looking at https://wiki.python.org/moin/WindowsCompilers

The author has installed the `MSVC` compiler with the Microsoft *Build Tools for Visual Studio* from:

https://visualstudio.microsoft.com/thank-you-downloading-visual-studio/?sku=BuildTools&rel=16 ,

following the instructions in

https://www.scivision.dev/python-windows-visual-c-14-required/ .

Before typing `python setup.py bdist_wheel` in a Windows command line the author had to type:

```
"C:\Program Files (x86)\Microsoft Visual Studio\2019\BuildTools\VC\Auxiliary\Build\
↪vcvars64.bat"
```

Here the path my be different on the user's Windows system.

2.5 The build process

Warning! At present the build process is under construction.

We plan to switch to the `Meson` build system. The reason for this is that invoking `setup.py` from the command line is deprecated. `Meson` requires a much more declarative style for describing the build operations. Thus the description here is pretty much outdated!

The standard build process for python packages is based on the `setuptools` package. We use `setuptools` to build the `mmgroup` package. We have added the following functionality to the standard build process:

- The user may call arbitrary functions or processes, e.g. for the automatic generation of C code.
- The user may add a shared library that may be used by several extensions.

Module `build_ext_steps` in the root directory contains the python classes required for that extension. The following table lists some important modules in the root directory.

Table 4: Files in the root directory used by the build process

File name	Purpose
`build_ext_steps.py`	Extension for the `setuptools` package
`build_shared.py`	Tool for building a shared library
`cleanup.py`	Clean up intermediate files
`config.py`	A configuration file of the project
`generate_code.py`	Tool for generating C code
`MANIFEST.in`	List of files to be added to the source distribution
`pyproject.toml`	Main configuration file for the project
`README.rst`	Main documentation file for GitHub
`setup.py`	Main file for the build process with `setuptools`

2.5.1 The tool build_shared.py

The tool `build_shared.py` is a simple stand-alone tool for generating a shared library with a standard C compiler. It supports Windows, Linux and macOS. This job could also be done with a tool like `CMake` or `Meson`; but for the time being it has been easier to extract this functionality from an older tool with a similar functionality.

Here are the options of the tool `build_shared.py`:

```
-h, --help            show this help message and exit
--name MAME           Set NAME of library to be generated. NAME should
                      have no extension and no prefix such as 'lib'.
--sources [SOURCES ...]
                      Add list SOURCES of to list of source files. Each
                      source file should be a .c file with extension '.c'.
--source-dir DIR      Set root directory DIR for all source files.
--include-path [PATHS ...]
                      Set list of PATHS for finding .h files to be used.
--library-path [PATHS ...]
                      Set list of PATHS for finding libraries to be used.
--define [VARS ...]   Define variables given by the list VARS for the
                      compiler. Each variable must be a string VAR or
                      VAR=VALUE.
--undef [VARS ...]    Undefine variables given by the list VARS for the
                      compiler.
--libraries [LIBS ...]
                      Search libraries with names in the list given by
                      LIBS. This corresponds to the gcc option '-l'.
--library-dir DIR     Set directory DIR for storing object files and
                      static libraries.
--shared-dir DIR      Set directory DIR for storing shared libraries.
--static [STATIC]     Create static instead of shared library. Optional
                      argument STATIC may be 0 (=shared) or 1 (=static).
--n N                 Use N parallel processes for compiling.
--compiler C          Specify name of default compiler. C must be 'unix',
                      'msvc', or 'mingw32'.
--cflags FLAGS        Add extra arguments FLAGS for the compiler. E.g.
                      '--cflags=-c,foo=bar' adds arguments '-c' and
                      'foo=bar'.
--display MODE        display (full) name of generated library and exit.
```

(continues on next page)

(continued from previous page)

```
                        MODE = 'load' displays the full file name of the
                        library. MODE = 'build' displays the name of the
                        library to be linked to a program using that
                        library.
```

2.5.2 Creating a new version

The ultimate goal of building a new version is to upload a new python version to the **pypi** server.

At present we will upload a source distribution and python 3 wheels for 64-bit Windows, Linux and macOS, say, for the latest two or three python versions.

Before creating a new version, (at least) the following test should be executed in a shell in Windows 64, and also in some Linux version:

```
pytest src/mmgroup/ -v -s -m "very_slow"
```

Version numbering

We assume that we want to create Version 0.0.8 at date 2022-07-12, with the short version description 'Performance improved'.

You should update the version number in file **setup.py** by writing e.g:

```
VERSION = '0.0.8' # 2022-07-12. Performance improved
```

into that file. You should also comment out older version descriptions in that file. In section **Version history** of file **docs/source/api.rst** you should add the following line to the version history:

```
| Version 0.0.8, 2022-07-12. Performance improved
```

Then you should upload the new version with these changes to the `master` branch in the github repository:

https://github.com/Martin-Seysen/mmgroup

After uploading, you should create a new release in the github repository. Therefore, click **Create a new release** in main page of the github repository. Here you should write the tag **v0.0.8** into the field **Choose a tag**. The **Target** of the release should be **master**, referring to the master branch in git. You should enter the title **mmgroup v0.0.8** into the field **Release title**. We recommend to enter (at least) the version description (which is 'Performance improved' in our case) into the field **Describe this release**. Finally, you should click the button **Publish release** in the github window.

Generating the wheels

This subsection describes how to create wheels manually. This process has now been automated to some extent by using `GitHub` actions that trigger the python tool `cibuildwheel`. So the reader may skip this section.

Here you must generate a wheel for each python version, and also for each operating system that you want to support. Here we assume that Anaconda is used for creating wheels for Windows 64 for various python versions. An Anaconda environment e.g for python 3.9 is created by typing the following command

```
conda create --name python39 python = 3.9
```

Then we may switch to that python version by typing:

```
conda activate python39
```

Environments for other python versions are created similarly. One has to install all required python packages for each version. For uploading a version to **pypi** we also have to install **twine** with

```
pip install twine
```

In each version to be supported we have to type:

```
python setup.py build_ext bdist_wheel
python setup.py sdist
```

Here the first line creates the wheel for the selected python version. Before doing so in Windows, you must install a C compiler for cython as described in section *Installation from a source distribution*.

The second line creates a source distribution; this must be done only once. The wheels and source distributions are stored in subdirectory **dist**. The wheel for mmgroup version 0.0.8 for python 3.9 has a name similar to `mmgroup-0.0.9-cp37-cp37m-win_amd64.whl`; and the source distribution has a name like `mmgroup-0.0.8.zip`.

Uploading the version to pypi

You may upload the new version with the following commend:

```
twine upload twine upload dist/*
```

This uploads all files from subdirectory **dist** to **pypi**. So you'd better cleanup that directory before uploading.

Warning: Uploading with **twine** is irreversible. If your uploaded version is buggy, you will have to create a new version!

2.5.3 Description of module `build_ext_steps.py`

Module `build_ext_steps` provides a customized version of the 'build_ext' command for setup.py.

It should be placed in the same directory as the module `setup.py`.

Distributing python packages

The standard toolkit for distributing python packages is the `setuptools` package. Here the user types:

```
python setup.py build_ext
```

at the console for building the extensions to the python package, which are typically written in a language like C or C++ for the sake of speed. We may use e.g. the `Cython` package to write python wrappers for the functions written in C or C++. The `setuptools` package supports the integration of `Cython` extensions.

The `setup.py` script describes a list of extensions, where each extension is an instance of class `Extension` which is provided by `setuptools`. Then it calls the `setup` function which builds all these extensions:

```
from setuptools import setup
from setuptools.extension import Extension

ext_modules = [
    Extension(
        ...  # description of first extension
    ),
    Extension(
        ...  # description of second extension
    ),
    ...
]

setup(
    ..., # Standard arguments for the setup function
    ext_modules = ext_modules,   # List of extensions to be built
)
```

We assume that the reader is familiar with the standard python setup process. For background, we refer to

https://setuptools.readthedocs.io/en/latest/

Added functionality for building python packages

This `build_ext_steps` module supports a new paradigm for building a python extension:

- A python program `make_stage1.py` creates a C program `stage1.c`.
- We create a python extension `stage1.so` (or `stage1.pyd` in Windows) that makes the functionality of `stage1.c` available in python.
- A python program `make_stage2.py` creates a C program `stage2.c`. Here `make_stage2.py` may import `stage1.so` (or `stage1.pyd`).
- We create a python extension that makes the functionality of `stage2.c` available in python.
- etc.

This paradigm is not supported by the `setuptools` package.

Using class `BuildExtCmd` for the new paradigm

For using the new building paradigm we have to replace the standard class `build_ext` by the class `build_ext_steps.BuildExtCmd`.

```
from setuptools import setup
from build_ext_steps import Extension
from build_ext_steps import BuildExtCmd

ext_modules = [
    # description of extension as above
]

setup(
    ..., # Standard arguments for the setup function
```

(continues on next page)

(continued from previous page)

```
    ext_modules = ext_modules,   # List of extensions to be built
    cmdclass={
       'build_ext': BuildExtCmd, # replace class for build_ext
    },
)
```

This change has a few consequences:

- It is guaranteed that the extension are build in the given order
- Extensions are always build in place (option `build_ext --inplace`) (The current version does not support building the extension in a special build directory.)
- The building of all extensions is now forced (option `build_ext --f`), regardless of any time stamps.

Apart from these changes, an extension is created in the same way as with `setuptools`.

For a documentation of the `Extension` class in the `setuptools` package, see

https://docs.python.org/3/distutils/apiref.html?highlight=extension#distutils.core.Extension

Inserting user-defined functions into the build process

Module `build_ext_steps` provides a class `CustomBuildStep` for adding user-defined functions to the build process.

In the list `ext_modules` of extensions, instances of class `CustomBuildStep` may be mixed with instances of class `Extension`, Class `CustomBuildStep` models an arbitrary sequence of functions to be executed.

The constructor for that class takes a string 'name' describing the action of these functions, followed by an arbitrary number of lists, where each list describes a function to be executed.

Here the first entry should be a string. Then a subprocess with that name is called. Subsequent entries in the list are arguments passed to the subprocess.

If such an argument contains the string '${build_lib}' then that string is replaced by the name of the root directory of the path where the extionsion is to be built. If the 'build_ext' command of `setup.py` is invoked with the '–inplace' option then the string '${build_lib}' is replaced by 'null'.

Such a subprocess may be e.g. a step that generates C code to be used for building a subsequent python extension.

Its recommended to use the string `sys.executable` (provided by the `sys` package) instead of the string `'python'` for starting a python subprocess.

The following functionality is deprecated:

If the first entry of a list as descibed above is a python function then that function is called. Subsequent entries in the list are passed as arguments to the function.

Using a shared library in an extension

The user may have to perform some os-specific steps for making the library available for python extension. Details are out of the scope of this documentation.

Using an extension in a subsequent build step

Once a python extension has been built, it can also be used in a subsequent step of the build process, e.g. for calculating large arrays of constants for C programs. Details are out of the scope of this documentation..

2.6 Code generation

Warning! At present the code generation process is under construction.

We plan to switch to the `Meson` build system. Therefore a much more declarative style is required for describing the code generation. Also, a strict separation between input and output directories is required. Thus the description here is pretty much outdated!

2.6.1 The main code generation tool *generate_code.py*

An invocation of the main code generation script *generate_code.py* generates a set of .c files and also a single .h file from a set of source files. Here each .c file is generated from a single source file which has extension .ske. Prototypes for the functions in the .c files may also be generated form these source files; they are copied into the generated .h file. The user may also specify a source file with an extension .h. Such a .h file usually contains global declarations; and it is also copied into the generated .h file.

The user may also specifiy a set of python modules to be used for code generation with the `--tables` option. Any such module must have a class `Tables` containing information how to enter tables and code snippets into the generated C code. The structure of such a code-generating module is described in the next section.

We use the *Cython* language to integrate the generated .c file into our python project. In a Cython project, a .pxd file should be generated from a header file. Here the .pxd file contains essentially the same information as the header file. The `--pxd` option generates a .pxd file from the generated header file automatically. Here the user may specify a source file (with extension .pxd), which will also be copied into the generated .pxd file.

The user may also create a .pxi file from the .pxd file with the `--pxi` option . Such a .pxi file will contains wrappers for the C functions declared in the .pxi file. These wrappers can be used directly from python by including the .pxi file into a .pyx file that defines a Cython extension. Note that this automatic wrappping mechanism works for rather simple prototypes only.

With the `--pyx` option, the user may simply copy a .pyx file from the source path to a destination directory.

The Meson build system requires strict separation between the input and the output directory. The user may specify a path where to look for the input files (with extensions .ske, .h, .pxd, and .pyx). The user may specify a directory where to store the generated .c and .h files; and also a directory where to store the generated .pxd, .pyi, and .pyx files.

In the following table we list the options available in the *generate_code.py* tool.

```
-h, --help            show this help message and exit
--sources [SOURCE ...]
                      List of SOURCE files to be generated. For each
                      SOURCE with extension '.c' a '.c' file is
                      generated from a file with the same name and
                      extension '.ske'. A SOURCE with extension '.h' is
                      copied into the common header file. Each SOURCE
                      is seached in the path set by parameter '--
                      source-path'. Output is written to the directory
                      set by parameter '--out-dir'.
--out-header HEADER   Set name of output header file to HEADER. By
```

(continues on next page)

(continued from previous page)

```
                      default we take the first file with extension .h
                      in the list given in the argument '--sources'.
--pxd PXD             Set input '.pxd' file PXD for generating '.pxd'
                      file with same name from that input file and from
                      the generated header.
--pxi                 Generate '.pxi' file from '.pxd' file if this
                      option is set.
--pyx PYX             Copy input '.pyx' file PYX from source path to
                      output directory.
--tables [TABLES ...]
                      Add list TABLES of table classes to the tables to
                      be used for code generation.
--set VAR=VALUE [VAR=VALUE ...]
                      Set variable VAR to value VALUE. When generating
                      code with subsequent '--sources' options then the
                      table classes will set VAR=VALUE.
--subst [PATTERN SUBSTITUTION ...]
                      Map the name of a '.c' or '.h' file to be
                      generated to the name of a file used as a source
                      for the generation process. Example: "--subst
                      mm(?P<p>[0-9]+)_op mm_op" maps e.g.'mm3_op' to
                      'mm_op'. We substitute PATTERN in the name of a
                      generated file by SUBSTITUTION for obtaining the
                      name of that source. The part "(?P<p>[0-9]+)"
                      creates a variable 'p' that takes the decimal
                      string following the intial letters 'mm' in the
                      file name. Then variable 'p' will be passed to
                      the table classes used for code generation.
                      PATTERN must be given in python regular
                      expression syntax.
--source-path [PATHS ...]
                      Set list of PATHS for finding source files to be
                      used for generation process.
--py-path [PATHS ...]
                      Set list of PATHS for finding python scripts
--out-dir DIR         Store output files with extensions .c and .h in
                      directory DIR.
--out-pxd-dir DIR     Store output files with extensions .pxd, .pxi,
                      and .pyx in directory DIR.
--dll DLL_NAME        Generate code for exporting C functions to a DLL
                      or to a shared library with name DLL_NAME.
                      Parameter DLL_NAME must be the same for all
                      generated C files to be placed into the same DLL.
                      This parameter is not used for any other
                      purposes.
--nogil               Optional, declare functions from .pxi file as
                      'nogil' when set.
--mockup              Use tables and directives for Sphinx mockup if
                      present
-v, --verbose         Verbose operation
```

2.6.2 Python modules used for code generation

In the main code generation tool *generate_code.py* the user may specify a list of python modules to be used for code generation. Any such module must have a class `Tables` that contains two dictionaries `tables` and `directives` to be used for code generation. Both of these dictionaries map identifiers to objects implementing tables or directives to be used in the code generation process. Here the identifiers should be valid python identifiers beginning with an alphabetic character. If several code generation modules are specified then the corresponding dictionaries in the `Tables` classes of these modules are merged into a single dictionary.

Dictionary `tables` maps identifiers to values. If we have e.g. `tables[P] = 3` then the expression `%{P}` in a source file with extension .ske evaluates to the string `3` in the generated .c file. More elaborated use cases for the `tables` dictionary are given in the following sections. A value in the `tables` dictionary may also be a list of integers; then there ia a way to to initialize an array of integers in the generated .c files with these integer values.

Dictionary `directives` maps identifiers to directives. Invoking such a directive in a source file causes the code generator to put a code snippet into the generated .c file. Here a typical use case is the multiplication of a part of a vector of the representation of the Monster group with a fixed matrix.

A class `Tables` as described above should accept arbitrary keyword arguments in the constructor. (A standard way to do this is to provide a parameter `**kwds` in the constructor.) The *generate_code.py* tool may pass such keyword arguments to the constructor of a class `Tables` either with the `--set` option or with the `--subst` option. Using the `--set` option is straightforward. Using `--subst` is a bit more involved. The following paragraph provides an example for using the `--subst` option.

The source file `mm_op_pi.ske` supports certain oparations of the Monster group on vectors modulo several small odd numbers `p`. The module `mm_op_pi.py` provides a class `Tables` that supports the generation of code snippets to be used in these operations. From the source file `mm_op_word.ske` we want to generate files `mm3_op_pi.c`, `mm7_op_pi.c`, … etc. for the operation modulo 3, 7, …, etc. In the corresponding code generation process there is an option

```
--subst mm(?P<p>[0-9]+)_op mm_op
```

that primarily describes a subtitution of strings that maps both, `mm3_op_pi`, and `mm7_op_pi` to `mm_op_pi`. So this substitution maps the name of a generated .c file to the name of a .ske file (without extension) from which that .c file is to be generated. The first argument `mm(?P<p>[0-9]+)_op` of the `--subst` option is the pattern to be substituted given in the standard python regular expression syntax. Note that the part `(?P<p>[0-9]+)` of that argument also creates a dictionary `{'p' : '3'}` mapping identifier ‘p’ to the string ‘3’, when we substitute the name `mm3_op_pi` using this pattern. When such a dictionary is created, the code generation tool passes the keyword argument `p = '3'` to the constructor of every class `Tables` used for the generation of file `mm3_op_pi.c`. This way the the `Tables` classes can be instructed to generate code snippets for operation modulo 3.

2.6.3 Generation of code snippets and tables

In this section we describe the most important functions and classes used for the automatic generation of C code.

The C code generator generates code from a source file automatically.

Class `TableGenerator` can be used to generate a .c and a .h file from *source* file. Here such a source file usually has the extension .ske.

Basic idea: We copy a *source* file to the .c file. The source contains directives to enter precomputed tables, code snippets or constants into the .c file, and prototypes into the .h file.

We copy the content of the *source* file directly to the .c file. In the *source* file we also parse certain directives and we replace them by automatically generated code snippets in the .c file and in the .h file.

The arguments given to the constructor of class `TableGenerator` specify the meaning of the directives.

Overview

In the *source* file we parse directives of the shape:

```
// %%<keyword>  <arg1>, <arg2>, ...
```

Each directive executes certain function associated to the <keyword>. There are built-in and user-defined directives. Built-in directives are listed in section *Built in directives*. User-defined directives are passed to the constructor in a dictionary `directives` with entries:

```
<keyword> : <directive_function>.
```

If a directive is found in the source file we call:

```
<directive_function>(*<all evaluated arguments>),
```

and the string returned by that function is merged into the generated .c file. The arguments of a directive must be given in python syntax and they are evaluated to python objects using some safe version of the python `eval()` function. Basically, python identifiers in an argument are evaluated as follows:

In the constructor, parameter `tables` is a dictionary of local variables, which will be passed to the python `eval()` function for evaluating an argument. See section *Arguments of directives* for a more detailed description of this process.

You may also consider class `UserDirective` for constructing specific directive functions.

Arguments of directives

The comma-separated list of arguments of a built-in or user-defined directive or function must be a valid python expression. This expression is evaluated by a safe version of the python `eval()` function, with the local variables given by the dictionary `names`. Here `names` is an updated version of the dictionary `tables` of local variables passed in the constructor of this class as described below. This means that each identifier found in the list of arguments is looked up in that dictionary, and that the identifier is evaluated to the corresponding value in that dictionary.

The dynamic dictionary `names` is initialized with dictionary `tables`. It may be updated with the names of the tables generated by the built-in directive `TABLE`, as indicated in the description of the built-in directive `USE_TABLE`.

There are a few more predefined entries in dictionary `names`:

name	value
`TABLES`	The original dictionary `tables` passed in the constructor
`NAMES`	updated version of dictionary `tables` as described above

Evaluating a python expression with the standard function `eval()` is known to be unsafe. For evaluating arguments we use a modified version of the `eval()` function. That modified function does not allow assignment, and identifiers starting with an underscore _ are illegal. See function `EvalNodeVisitor` in module `mmgroup.generate_c.generate_functions`. for details.

There is a limited number of built-in python functions (specified in directory `safe_locals` in the same module) that are legal in python expressions for arguments:

```
abs, int, len, max, min, str, pow, range, zip
```

An example with user-defined directives is given in class `testcode.TestGen`. We recommend to follow the conventions in that class for passing user-defined directives to an instance of class `TableGenerator`.

Built-in directives

- **COMMENT, deprecated!**

  Indicates that a comment follows. This should be used only for comments relevant for the user, not for internal details. No action in this class, but other functions may use this directive for generating user documentation.

- **ELSE [IF]? <expression>**

  else clause for IF directive, see IF

- **END <directive>**

  Closes a block that has been started by one of the directives FOR or IF. A FOR block must be closed with END FOR, an IF block must be closed with END IF, etc.

- **ERROR <message>**

  TODO: yet to be implemented and documented!!!!

- **EXPORT (arguments see below)**

  Then the following line with an appended semicolon is copied into the generated .h file. This is useful for creating prototypes. The EXPORT directive may precede a line with a USE_TABLE directive. Then the table name captured by the next USE_TABLE statement is exported to the generated .h file. The EXPORT directive may precede a line with a DEFINE function. Then the #define statement generated by the next DEFINE function is written to the generated .h file.

  An optional argument p means that the exported line will also be included in a .pxd file, see method generate_pxd().

  An optional argument px or xp means that the same as argument p. Then in addition we also write a comment # PYX <wrap pxi> into the .pxd file generated by method generate_pxd(). If function mmgroup.generate_c.pxd_to_pyx reads such a comment in a .pxd file, it will write a simple wrapper for the exported function into a .pyx file.

- **EXPORT_KWD <keyword>**

  The directive:

  ```
  // %% EXPORT_KWD  FOO_API
  ```

  places the keyword FOO_API before any function or variable which is exported with the EXPORT or EXPORT_TABLE directive. This is useful on a MS Windows system for creating a dll. Then the header should contain a definition similar to:

  ```
  #define FOO_API __declspec(dllexport)
  ```

  This declares these functions or variables to be exported by the dll. For background, see e.g. https://gcc.gnu.org/wiki/Visibility.

- **EXPORT_TABLE (no arguments)**

  Parses the next input line for the C-name of a table, and exports this name into the generate .h header file. This is equivalent to two subsequent directives:

  ```
  EXPORT
  USE_TABLE
  ```

  There are, however, subtle differences, see directive EXPORT_KWD.

- **FOR <variable> in <parameter_list>**

  Generate a sequence of blocks of statements, one block for each parameter in the parameter_list. A sequence of arbitrary lines follows, terminated by an END FOR directive. The parameter list is evaluated to a python object which should be iterable.

Nesting of FOR and similar directives is possible.

Examples:

```
// %%FOR x in range(1,3)
i%{x} += 2*%{x};
// %%END FOR
```

evaluates to:

```
i1 += 2*1;
i2 += 2*2;
```

- **GEN <ch>**
  Directs output to .c or to .h file or to both, depending on whether the letter c or h is present in the first argument.
- **IF <expression>**
  Conditional code generation depending on the boolean value of <expression>. Syntax is:

  ```
  // %%IF <expression>
  <statements>
  // %%ELSE IF <expression>
  <statements>
  // %%ELSE
  <statements>
  // %%END IF
  ```

  with one or more optional `ELSE IF` clauses and at most one final `ELSE` clause. Nested `IF` and `FOR` blocks are possible.
- `INCLUDE_HEADERS`

  This directive is legal in a header file with extension `.h` only. A typical header file may look like this:

  ```
  #ifndef  _THIS_HAEDER_HAS_ALREADY_BEEN_PROCESSED_
  #define  _THIS_HAEDER_HAS_ALREADY_BEEN_PROCESSED_

  // Some prototypes for C functions will be inserted here

  #endif
  ```

  The code generator may insert automatically generated prototypes for C functions into that header. You can tell the code generator where to insert these prototypes as follows:

  ```
  #ifndef  _THIS_HAEDER_HAS_ALREADY_BEEN_PROCESSED_
  #define  _THIS_HAEDER_HAS_ALREADY_BEEN_PROCESSED_

  // %%INCLUDE_HEADERS

  #endif
  ```
- **JOIN <infix>, <suffix>**
  A `FOR` directive must follow a `JOIN` directive. The expressions generated by the following `FOR` directive are joined with the <infix> (which is usually an operator), and the <suffix> is appended to the whole generated expression. E.g.:

```
a +=
// %%JOIN*  " +", ";"
// %%FOR* i in range(1,4)
  %{int:2*i} * table[%{i}]
// %%END FOR
```

evaluates to:

```
a +=
  2 * table[1] +
  4 * table[2] +
  6 * table[3];
```

- **PY_DOCSTR <string>, <format>**

  This directive is no longer supported

- **PYX <string>**

  This directive is no longer supported

  into the .pxd file, when generating a .pxd file. This has no effect on the generated .c or .h file. It may be used for automatically generating a .pyx file from a .pxd file.

- **TABLE <table>, <format>**

  Enters all entries of the python array <table> into the C code as an array of constants with an optional <format>. Only the data of the array are written. Feasible <format> strings are given in function format_number().

- **USE_TABLE (no arguments)**

  Parses the next input line for the C-name of a table and makes that C name available for user-defined directives.

  If a table is coded with the `TABLE` directive and preceded by a `USE_TABLE` or `EXPORT_TABLE` directive then the name of that table will be added to the dictionary `names`. The key with the python name of that table will have the C name of that table as its value.

  Consider the following example:

  ```
  1:    // %%USE_TABLE
  2:    int  C_table[]  = {
  3:       // %%TABLE python_table, uint32
  4:    }
  ```

  Here `'python_table'` must be a key in the dictionary `tables` passed to the constructor of class `TableGenerator`. The value for that key should be a list of 32-bit integers. Then the directive in line 3 creates the list of integer constants given by the list `tables[python_table]`.

  The `USE_TABLE` directive in line 1 has the following effect. In the sequel the entry `names['python_table']` has the value `'C_table'`, which is the C name of the table that has been generated. So we may use that entry in subsequent directives or via string formatting with curly braces.

  The entries of table `python_table` are still available in the form `python_table[i]` as before. `python_table[:]` evaluates to the whole table as before.

- **WITH <variable> = <value>**

  Temporarily assign <value> to <variable>. A sequence of arbitrary lines follows, terminated by an `END WITH` directive. Both, <variable> and <value>, may be tuples of equal length; a nested tuple <variable>

is illegal. The <variable>, or the variables contained in the tuple <variable>, are temporarily added to the dictionary `names` as keys, to that they can be used via string formatting.

Nesting of WITH and other directives is possible.

Example:

```
// %WITH square, cube = int((x+1)**2), int((x+1)**3)
i += %{cube} + %{square} + %{x};
// %%END WITH
```

Assuming `x` has value `9`, this evaluates to:

```
i += 1000 + 100 + 9;
```

String formatting

A line `l` in the source file may contain a string `s` with balanced curly braces and preceded by a '%' character. Then the standard `.format()` method is applied to the string `s[1:]` obtained from `s` by dropping the initial '%' character. The dictionary `names` is given to the formatting method as given as the list of keyword arguments.

E.g. in case `names['x'] = y`, the expressions `%{x.z}` and `%{x[z]}` refer to `y.z` and `y[z]`. Of course, the objects `y.z` or `y[z]` must exist and evaluate to strings which are meaningful in the generated C file.

For formatting, the dictionary `names` is an updated version of the dictionary `tables` passed in the constructor, as described in section *Arguments of directives*.

We consider this to be a reasonable compromise between the standard C syntax and the very powerful pythonic `.format` method for strings.

Using functions via string formatting

User-defined functions can be invoked inside C code via the the string formatting operator `%{function_name:arg1, arg2,...}`. Then the function with name `function_name` is called with the given arguments, and the result of the function is substituted in the C code.

Such a user-defined function may be created with class `UserFormat`. The simplest way to create such a user-defined function from a function `f` is to create an entry:

```
"function_name" : UserFormat(f)
```

in the dictionary `tables` passed to the constructor. Then `%{function_name:arg1,arg2,...}` evaluates to `str(f(arg1,arg2,...))`. Here arguments `arg1, arg2` must be given in python syntax and they are processed in the same way as the arguments of a directive described in section *Arguments of directives*. Then you may also write `function_name(arg1,..)` inside an argument of a directive or a user-defined function; this expression evaluates to `f(arg1,...)`.

Class `UserFormat` also contains options to control the conversion of the result of function f to a string.

There are a few built-in string formatting functions, as given by dictionary `built_in_formats` in module `generate_functions`.

There are a few predefined functions for string formatting:

- `%{int:x}` returns `x` as a decimal integer
- `%{hex:x}` returns `x` as a hexadecimal integer
- `%{len:l}` returns length of list `l` as a decimal int

- `%{str:x}` returns `str(x)`
- `%{join:s,l}` returns `s.join(l)` for string `s` and list `l`

See dictionary `built_in_formats` in module `mmgroup.generate_c.generate_functions` for details.

So, assuming that keys `'a'` and `'b'` have values `1` and `10` in the dictionary `tables` passed to the code generator, we may write e.g.:

```
y%{a} += %{int:3*b+1};
```

in the source file. This evaluates to:

```
y1 += 31;
```

Quiet form of a directive

For directives such as `IF`, `FOR`, ... there are also quiet variants `IF*`, `FOR*`, ..., which perform the same operations, but without writing any any comments about the directive into the output file. E.g. in:

```
// %%FOR i in range(50)
printf("%d\n", i);
// %%IF* i % 7 == 0 or i % 10 == 7
printf("seven\n");
// %%END IF
// %%END FOR
```

you probably do not want any messy comments about the IF directive in your generated code.

Historical remark

In versions up to 0.4 string formatting operator was `{xxx}` instead of `%{xxx}`. This had the unpleasent effect that valid C expressions have a different meaning in the input and and the output of the code generation process. This situation became unbearable after intoducing doxygen for documentation of C code. Then expressions like `\f$ x_{1,1} \f$` could not be coded in a reasonable way the old version.

2.6.4 Classes and functions provided by the code generator

class `mmgroup.generate_c.`**`TableGenerator`**(*tables={}*, *directives={}*, *verbose=False*)

Automatically generate a .c file and a .h file from a *source*

Basic idea: We copy a *source* file to the .c file. The source contains directives to enter precomputed tables, code snippets or constants into the .c file, and prototypes into the .h file.

We copy the content of the *source* file directly to the .c file. In the *source* file we also parse certain directives and we replace them by automatically generated code snippets in the .c file and in the .h file.

The arguments given to the constructor of this class specify the meaning of the directives.

For a more detailed description of these directives see module `make_c_tables_doc.py`

Parameters

- **`tables`** – Here `tables` is a dictionary of with entries of shape:

```
``name`` : ``table`` .
```

name is a string that is a valid python identifier. table is the python object to be referred by that name whenever an argument is evaluated as a python expression.

- **directives** – Here directives is a dictionary of shape:

```
``function_name`` : ``function``  .
```

The function function is executed when a directive of shape:

```
// %%function_name
```

occurs as a directive in the *source* file. The arguments following the directive are evaluated and passed to the function function. That function should return a string. This string is merged into the output .c file.

If directives is mmgroups.generate_c.NoDirectives then also built-in directives are not executed.

- **verbose** – optional, may be set True for verbose console output

External modules importing this class should use methods set_tables, generate, and table_size only.

generate(*source*, *c_stream*, *h_stream=None*, *gen='c'*)

Generate a .c and a .h file form a source.

Here source must be an iterator that yields the lines of the source file. This may be a readble object of class _io.TextIOWrapper as returned by the built-in function open().

Parameters c_stream and h_stream must be either None or an instance of a class with a write method, e.g. a writable object of class _io.TextIOWrapper. Then the output to the c file and to the h file is written to c_stream and to h_stream.

class mmgroup.generate_c.**UserDirective**(*f*, *eval_mode=''*, *prefix=False*)

Convert a function for use in a code generator directive

Each code generator directive corresponds to a function. Evaluating that function with the arguments given to the directive results in a string that makes up the code generated by the directive.

The list of arguments of a code generator directive is necessarily a string. Usually, this string is converted to a python expression which is evaluated using the dictionary ‘tables’ given to the code generator. Here dictionary tables contains the local variables used by the evaluation process.

The constructor of this class returns an object which is a callable function. Calling that function has the same effect as calling the function f, which is the first argument passed to the constructor.

Additional information for dealing with the arguments of the function may be given the constructor. This information is used by the code generator if an instance of this class is registered at the code generator as a directive. For registering a directive in that way you may simply enter an instance of this class as a value in the dictionary *directives* passed to the constructor of class TableGenerator.

Using this feature you may automatically convert arguments of the function e.g. to strings, integers or python objects. You may also prepend a local variable from dictionary tables to the list of user-specified arguments of the directive. The latter feature gives a directive the flavour of a member function of a class, similar to prepending self to the list of user-specified arguments.

Parameters

- **f** – A callable function which is executed when the code generator parses a user-defined directive. The arguments of such a directive are passed to function f as arguments.

- **eval_mode** – This is a string specifying the syntax of the arguments. The i-th character of the string `eval_mode` controls the evaluation of the i-th argument of function `f` as follows:
  - `'p'` Evaluate the argument to a python object (default).
  - `'i'` Evaluate the argument to an integer.
  - `'s'` Evaluate the argument to a string, ignoring dictionary

  It `eval_mode` is `'.'` then a single parameter containing just the given string is passed to function `f` as a single argument without any parsing with python.
- **prefix** – Describes to an optional local variable of the code generator prepended to the list of arguments of the directive. It may be one of the following:
  - `False`: Nothing is prepended
  - `True`: The dictionary of all local variables is prepended
  - any string: The local variable with that name is prepended

class `mmgroup.generate_c.`**`UserFormat`**(*f*, *eval_mode=''*, *prefix=False*, *fmt=None*, *arg_fmt=None*)

Convert a function for string formatting in the code generator

The code generator evaluates expressions in curly braces `%{...}` in the source code with the python `.format()` function for stings.

It is desirable that `%{int:2*3}` evaluates to a string representing the integer constant `int(2*3)` for some built-in and user-defined functions such as function `int`.

Therefore we need a python object with name `int` that behaves like the function `int()` when being called. That object should also have a `__format__` method that returns `str(int(<expression>))` if a string representing a valid python expression is passed to the `__format__` method.

`UserFormat(int)` creates such an object. Thus:

```
tables = {"int": UserFormat(int)}
"{int:2*3}".format(**tables))
```

evaluates to `str(int(2*3))`. Of course, function `int` may be replaced by any user-defined function.

Things get more interesting if we prepare a dictionary for the code generator:

```
from mmgroup.generate_c import TableGenerator, UserFormat
# Prepare tables for a code generator
tables = {"int": UserFormat(int),  "a": 3, "b":5}
# Create a code generator 'codegen'
codegen = TableGenerator(tables)
# Make a sample source line of C code
sample_source = "   x = %{int: a * b + int('1')};\n"
# generate file "test.c" containing the evaluated sample line
codegen.generate(sample_source, "test.c")
```

Here the string `{int:  a * b + int('1')}` in the sample source line is formatted with the standard python `.format` method. The `.format` method obtains the entries of the dictionary `tables` as keyword arguments.

So `%{int:  a * b + int('1')}` evaluates to `str(3*5+1)`, and the C file `"test.c"` will contain the line:

```
x = 16;
```

So `UserFormat(f)` creates an object the behaves like `f` when it is called; and it returns `str(f(arg1,arg2,...))` when it is invoked in `"{f:arg1,arg2,...}".format(f = f)`.

Here the argument list `arg1, arg2,...` is processed in according to the same rules as the argument list of a user-defined directive in class `UserDirective`.

Parameters

- **f** – A callable function which is executed when the code generator parses a user-defined formatting string in curly braces preceded by a '%' character, such as `%{...}`. The arguments given in the formatting string are passed to function `f` as arguments.
- **eval_mode** – This is a string specifying the syntax of the arguments in the same way as in class `UserFunction`.
- **prefix** – Describes to an optional local variable of the code generator prepended to the list of arguments of the function in the same way as in class `UserFunction`.
- **fmt** – By default, function str() is applied to the result of function `f` in order to convert that result to a string. This behaviour can be modified by setting parameter `fmt`.

  if `fmt` is a callable function then `str(f(*args))` is replaced by `fmt(f(*args))`.
- **arg_fmt** – If this parameter is set then the corresponding `.format` method returns `arg_fmt(*args)` instead of `str(f(*args))`. This overwrites the parameter `fmt`.

`mmgroup.generate_c.`**`c_snippet`**(*source*, **args*, ***kwds*)

Return a C code snippet as a string from a *source* string

Here `source` is a string that is interpreted in the same way as the text in a source file in method `generate()` of class `TableGenerator`. The function applies the code generator to the string `source` and returns the generated C code as a string.

All subsequent keyword arguments are treated as a dictionary and they are passed to the code generator in class `TableGenerator` in the same way as parameter `tables` in the constructor of that class.

One can also pass positional arguments to this function. In the `source` string they an be accessed as `%{0}`, `%{1}`, etc.

A line starting with `// %%` is interpreted as a directive as in class `TableGenerator`.

The keyword `directives` is reserved for passing a dictionary of directives as in class `TableGenerator`.

In order to achieve a similar effect as generating C code with:

```
tg = TableGenerator(tables, directives)
tg.generate(source_file, c_file)
```

you may code e.g.:

```
c_string = c_snippet(source, directives=directives, **tables)
```

If this function cannot evaluate an expression of shape `%{xxx}` then the expression it is not changed; so a subsequent code generation step may evaluate that expression. An unevaluated argument in a directive leads to an error.

`mmgroup.generate_c.`**`format_item`**(*n*, *format=None*)

Format a python object `n` to a C string for use in a C table

The default format is to apply function str() to the object.

The current version supports integer formats only. Parameter `format` must be one of the following strings:

```
'int8', 'int16', 'int32', 'int64', 'uint8', 'uint16', 'uint32', 'uint64'
```

If format is one of `'int8', 'int16', 'int32', 'int64'` then an integer is reduced modulo the appropriate power of two and formatted hexadecimal. Everything else is formatted 'as is'. An argument `n = None` is interpreted as the NULL poiner in C.

`mmgroup.generate_c.`**`make_doc`**`()`

`mmgroup.generate_c.`**`make_table`**`(`*table*, *format=None*`)`

Convert a python table to a C sequence separated by commas

Parameter `table` may be anything iterable. Function

```
format_item(x, format)
```

is applied to each item `x` in the table.

The output may contain several lines. Iterated lists of integers (or strings) are converted to C constants as expected. E.g. the list `[[1,2], 3, [4,5,6]]` is converted to something like:

{1,2}, 3, {4,5,6}

`mmgroup.generate_c.`**`prepend_blanks`**`(`*string*, *length*`)`

Prepend `length` blank characters to each line of a `string`

Here parameter `string` is a string that may consist of several lines.

`mmgroup.generate_c.`**`pxd_to_pxi`**`(`*pxd_file*, *module=None*, *translate=None*, *nogil=False*`)`

Extract Cython wrappers from prototypes in a .pxd file

A .pxd file contains prototypes of external C functions and it may be included into a .pyx file in order to create a Cython module that uses these C functions.

This function returns a string containing python wrappers of the C functions in the pxd file. This string may be used as a part of an automatically generated .pxi file that can be included directly into a .pyx file. Here the C functions in the .pxd file to be included must be sufficiently simple as indicated below.

The returned string starts with two lines:

```
cimport cython
cimport <module>
```

Here <module> is given by the parameter `module`, By default, `module` is constructed from the name `pxd_file` of the . pxd file.

The .pxd file is parsed for lines of shape:

```
<return_type> <function>(<type1> <arg1>,  <type2> <arg2> , ...)
```

Every such line is converted to a string that codes function, which is a Cython wrapper of that function of shape:

```
@cython.boundscheck(False)  # Deactivate bounds checking
@cython.wraparound(False)   # Deactivate negative indexing.
def <translated_function>(<arg1>, <arg2> , ...):
    cdef <type1> <arg1> = <arg1>_v_
    cdef <type2> <arg2> = <arg2>_v
    cdef <return_type>  _ret
    ret_ = <module>.<function>(<arg1>_v_,  <arg2>_v_ , ...)
    return ret_
```

`<translated_function>` is the string computed as `translate(<function>)`, if the argument `translate` is given. Otherwise is `<translated_function>` is equal to `<function>`.

<return_type> and <type1>, <type2>, ... must be valid types known to Cython and C. The types <type1>, <type2> used for arguments may also be pointers. Then pointers are converted to memory views e.g:

```
uint32_t <function>(uint8_t *a)
```

is converted to:

```
@cython.boundscheck(False)  # Deactivate bounds checking
@cython.wraparound(False)   # Deactivate negative indexing.
def <translated_function1>(a):
    cdef uint8_t[::1] a_v_ = a
    cdef uint32_t ret_
    ret_ = <function>(&a_v_[0])
    return ret_
```

The <return_type> may be void, but not be a pointer. Other types are not allowed.

We convert a function only if a line containing the string "# PYX" (possibly with leading blanks) precedes the declaration of a function. In the code generator you may use the EXPORT directive with options px to enter such a line into the source file.

If nogil is True, a C function is called with as follows:

```
with nogil:
    ret_ = <function>(&a_v_[0])
```

Then the C function <function> must be e.g. declared as follows:

```
cdef extern from "<file>.h" nogil:
    int <function> (int a, int *b)
```

This feature releases the GIL (Global Interpreter Lock) when calling the function.

mmgroup.generate_c.**generate_pxd**(*pxd_out*, *h_in*, *pxd_in=None*, *h_name=None*, *nogil=False*)

Create a .pxd file from a C header file.

Here parameter h_in is the name of a C header file that has usually been generated with method generate of class TableGenerator. In such a header file, some exported function are preceded by a comment of shape

```
// %%EXPORT <parameter>
```

There <parameter> is a string of letters describing the way how a function is exported. The function is entered into the output .pxd file if <parameter> contains the letter 'p'.

Method generate_pxd creates a .pxd file with name pxd_file that contains information about these exported function.

A .pxd file is used in Cython. Its content is:

```
cdef extern from <h_file_name>:
    <exported function 1>
    <exported function 2>
    ...
```

Here <h_file_name> is given by parameter h_name.

Parameters

- **pxd_out** – Name of the output .pxd file.

- **h_in** – Name of the header file from which external functions are copied into the .pxd file as a `cdef extern from` statement.
- **pxd_in** – An optional name of a input .pxd file to be copied in the output .pxd file in front of the `cdef` statement. If that parmeter is a string containing a newline character `\n` then that string is copied directly into the output .pxd file.
- **h_name** – Name of .h file to to be placed into the statement `cdef extern from <h_file_name>`.
- **nogil** – Optional, exported functions are declared as `nogil` when set.

2.6.5 Support for Sphinx *mockup* feature when generating documentation

We use the *Sphinx* package on the *Read the Docs* server for generating documentation. On that server we do not want to compile .c files. However, we want the .c files to be present for extracting documentation from them, using the *Doxygen* tool. Here it suffices if the prototypes and comments relevant for documentation are present in the .c files.

This means that we have to perform the code generation process also on the documentation server. Note that some code generation procedures in the `Tables` classes in the python modules used for code generation rely on compiled Cython extensions. But Cython extensions will not be compiled on the documentation server. To avoid using compiled code, in each python module to be used for code generation we may add a `MockupTables` class to the `Tables` class already present. Such a `MockupTables` class may generate the parts of a .c file relevant for documentation only, without using any compiled Cython extensions.

We may set the `--mockup` option in the code generation tool *generate_code.py*. Then any `Tables` class in a python module to be used for code generation is replaced by the `MockupTables` class in that module, if present.

2.6.6 How the code generator is used

Warning!

At present the process of generating C code is under construction.

We plan to switch to the `meson` build system. Therefore a much more declarative style is required for describing an buld operations. Thus the description here is pertty much outdated!

For generating C code, an instance of class `TableGenerator` in module `mmgroup.generate_c` must be generated. That instance takes two arguments `tables` and `directives` as described in section *Code generation*.

There are many tables and directives needed for different purposes. By convention we provide a variety of classes, where each class provides two attributes `tables` and `directives` providing dictionaries of tables and directives, respectively. These dictionaries are suitable for creating instances of classes `TableGenerator`. Such classes are called *table-providing classes*. Attributes `tables` and `directives` may also be implemented as properties.

Python scripts with names `codegen_xxx.py` in the root directory of the source distribution create instances of class `TableGenerator` that will generate C programs and also the corresponding header files. These python scripts are run as subprocesses of the `build_ext` command in the `setup.py` script.

Each instance of class `TableGenerator` may take the union of the directories obtained from methods `tables()` and `directives()` of several table-providing classes. A simple example for a table-providing class is the class `Lsbit24Function` in module `mmgroup.dev.mat24.mat24_aux`.

2.7 Description of some typical bugs

In this section we decribe some typical bugs that may occur during the develpment process and that may be extremely hard to find.

2.7.1 1. Cleaning up a corrupted local git repository

After a long development session the local git repository might be corrupted for what reason ever. Here the best cure is to delete everything in the repository (exept for subdirectory .git) and to checkout everythinh with git. If you do not want to do so, you may cleanaup intermediate files as follows.

First open a command shell, switch to the root directory of your local repository, and make sure that your C compiler works in that shell. Then enter the following statements:

```
python cleanup.py -a
git checkout .
python setup.py  build_ext --inplace
python -m pytest ./src/mmgroup/ -v -s -m "not slow"
```

Then for creating the documentation you should enter something like the following (details depending on the operating system):

```
cd docs
sphinx-build -M html source build  -E -a
sphinx-build -M latexpdf source build  -E -a
cd ..
```

2.7.2 2. Documentation is created correctly on the local host but not on readthedocs

Whenever we update the *master* branch in *git* on the *gitbub* server, this automatically starts a process on the *readthedocs* server that updates the documentation. In that case the developer should check the documentation on the *readthedocs* server at

https://mmgroup.readthedocs.io/en/latest/index.html

If this documentation is incorrect then the developer should check the logfiles of the last generation of the documentation on *readthedocs*. For generating documentation, the *readthedocs* server starts a sequence of processes, and for each process the output is logged.

There is one type of bug that is difficult to find. If this bug occurs, then in the output of one of the processes that invokes `python -m sphinx ...` there may be an error message similar to the following message:

WARNING: autodoc: failed to import module 'mm_group' from module 'mmgroup'; the following exception was raised:

No module named 'mmgroup.mat24'

This message has to do with the mockup of python extensions when running the *Sphinx* documentation tool on the *readthedocs* server.

We do not use any C compiler on the readthedocs server, so that python extensions written in *Cython* are not available. When running *Sphinx* then all python objects to be documented are imported, so that their docstrings can be processed. Therefore all Cython extensions must be mocked up in *Sphinx*. This means that we tell *Sphinx* that these extensions are not available. For details see https://www.sphinx-doc.org/en/master/usage/extensions/autodoc.html#confval-autodoc_mock_imports

The Cython extension `mat24` is not mocked up in *Sphinx*, but it is replaced by class `mmgroup.dev.mat24.mat24_ref.Mat24` instead.

So instad of simply importing the extension `mat24` we have to do the following:

```
>>> try:
>>>    from mmgroup import mat24
>>> except (ImportError, ModuleNotFoundError):
>>>    from mmgroup.dev.mat24.mat24_ref import Mat24
>>>    mat24 = Mat24
```

More sophisticated examples of importing objects from module `mat24` are given e.g. in modules `mmgroup.mm_group` and `mmgroup.structures.autpl`.

One way to debug such a problem on the local host (and not on the *readthedocs* server) is as follows. Use `pip uninstall mmgroup` (or similar) to uninstall the `mmgroup package` Then execute the following commands in your shell:

```
python cleanup.py -a
git checkout .
set readthedocs=True
python setup.py  build_ext --inplace
cd docs
sphinx-build -M html source build  -E -a
sphinx-build -M latexpdf source build  -E -a
cd ..
# Here you should inspect the output of the sphinx-build
# commands for debugging
unset readthedocs  # in Windows use 'set readthedocs=' instead
# Reverse the effect of  'set readthedocs=True'
```

If you are lucky then the errors occuring on the *readthedocs* server will now also occur on your local host.

Warning: After typing that sequence into your shell, your environment is probably corrupted; and you should proceed as in subsection **Cleaning up a corrupted local git repository**.

CHAPTER

THREE

THE C INTERFACE OF THE MMGROUP PROJECT

3.1 Introduction

This *document* describes the functionality of the C modules in this project. For most of these C modules there is also a python extension. Unless otherwise stated, each documented C function is wrapped by a Cython function with the same name and signature.

Note that almost all parameters of such a C function are declared as (signed or unsigend) integers, or as pointers to such integers. In python, a `numpy` array of appropriate `dtype` may be passed as an argument to a parameter delared as a pointer to an integer.

3.2 Description of the `mmgroup.mat24` extension

The automatically generated file `mat24_functions.c` contains the C code for the functions exported by the python extension `mmgroup.mat24`.

These functions are documented in file `mat24_functions.c`. Here just give an overview of the functionality of that module.

File `mat24_functions.c` has been generated from the source file `mat24_functions.ske` using the code generator in module `mmgroup.generate_c`.

The functions in file `mat24_functions.c` perform basic computations in the Golay code, in its cocode, and in the Mathieu group `Mat24`. They also deal with Parker loop `Pl` and with its automorphism group `AutPl`. A more comfortable python interface to these objects is provided by the python classes `GCode`, `Cocode`, `PLoop`, and `AutPL` in module `mmgroup`.

All C functions in file `mat24_functions.c` start with the prefix `mat24_`. These functions are also called from C functions in other modules. Therefore we store the binary code for these functions in a shared library. For some of these functions there are also macros (defined with `#define`), starting with `mat24_def_`.

There is a one-to-one correspondence between the functions in `mat24_functions.c` and the function exported from the python extension `mmgroup.mat24`, see subsection *Mapping C functions to python functions* for details.

The python class `Mat24` in module `mmgroup.dev.mat24.mat24_ref` contains pure python implementations of most functions of the `mmgroup.mat24` extension as class methods. This class is used for testing and as a substitute for the `mmgroup.mat24` extension in an early stage of the build process.

In the following subsections the term *C functions* refers to the C functions in file `mat24_functions.c`. In the documentation, the names of the C functions are given without the `mat24_` prefix. The term *API reference* means the main document *The mmgroup API reference* of this project.

3.2.1 The Golay code `C` and its cocode `C*`

The Mathieu group `Mat24` operates as a permutation group on a set of 24 elements which we label with numbers 0,. . . ,23 for use in Python and C. So it also operates on a vector space `V = GF(2)**24`, with `GF(2) = {0,1}`. Here `**` means exponentiation.

A vector `v` in a vector space over `GF(2)` is called a bit vector. We represent a bit vector as an integer, so that the `i`-th bit of `v` (with valence `2**i`) is the `i`-th component of `v`.

The Golay code `C` ia a 12-dimensional subspace of `V` fixed by `Mat24`. There are functions for checking and completing codewords and for getting the syndrome of a 24-bit vector.

We internally use a basis of `V` such that the first 12 basis vectors are a transversal of the Golay cocode and the last 12 basis vectors span the Golay code. These basis vectors are listed in the API reference.

We represent vectors in `V`, `C` and `C*` and in the subset of octads of `C` as follows:

The 759 octads are numbered from 0 to 758. They do not form a vector space. The 2**12 Golay code words are represented as binary numbers 0 to 4095. The 2**12 cocode words are represented as binary numbers 0 to 4095. A more detailed description is given in the API reference.

As usual, binary numbers representing bit vectors are added with the XOR operation `^`. Unused high bits in input bit vectors are ignored.

Functions changing an object from one representation `xxx` to another representation `yyy` are named `xxx_to_yyy`, where `xxx`, `yyy` is as follows:

```
vect:      standard representation of a bit vector in V = GF(2)**24
           coded as a 24-bit integer.
vintern:   internal representation a bit vector in V as a vector in
           the basis given above, coded as 24-bit integer.
gcode:     representation of a Golay code word in the basis given
           given above, coded as 12-bit integer.
octad:     representation as an octad numbered from 0 to 758
           (in lexical order given by representation  'gcode')
cocode:    representation as a cocode word in the basis given above,
           coded as 12-bit integer.
```

All these representations are given as unsigned integers.

We implement the following conversion functions:

```
vect_to_vintern, vintern_to_vect, vect_to_cocode,
vintern_to_vect, gcode_to_vect, cocode_to_vect.
```

Here irrelevant bits of the input are ignored. Function `cocode_to_vect` returns one of many possible solutions.

In the following functions the input is checked and the function fails in case of an error:

```
vect_to_gcode, vect_to_octad, gcode_to_octad,
octad_to_vect, octad_to_gcode
```

In case of failure, these C functions return a special value as indicated in the documentation of the function in the .c file. The corresponding python functions raise `ValueError` in case of failure.

Function `syndrome()` takes a vector `v` and calculates its syndrome, which is a vector of minimum weight equivalent to `v` modulo the Golay code. Function `cocode_syndrome()` takes a `cocode` representation of a cocode word instead.

Function `scalar_prod()` returns the scalar product of a Golay code vector in `gcode` and a cocode vector in `cocode` representation.

3.2.2 The Mathieu group Mat24

This class also contains support for the Mathieu group `Mat24`. An element of `Mat24` can be represented in one of the following ways:

```
perm:    Representation as an array of length 24 encoding a
         permutation of the integers 0,...,23 as a mapping.

m24num:  Representation as an integer 0 <= i < 244823040. Here i
         is the number of the permutation in lexicographic order.
         So the identity permutation is coded as 0.

matrix:  Representation as a 12 x 12 bit matrix acting on the Golay
         code by right multiplication. This matrix acts on a Golay
         code vectors (given in the 'gcode' representation) by
         right multiplication.
         Such a matrix is implemented as an array of integers with
         each integer corresponding to a row vector of the matrix.
         The purpose of this representation is to support
         the Parker loop and its automorphism group. Therefore a
         row vector is implemented as a 32-bit integer.
```

We implement the following conversion functions:

```
m24num_to_perm, perm_to_m24num, perm_to_matrix, matrix_to_perm.
```

There is a function `perm_check()` for checking if an array of length 24 really represents an element of the Mathieu group `Mat24`. All other function operating on `Mat24` in any way do not check if their inputs are really in `Mat24`. They will output garbage on bad input, but they are not supposed to crash.

The easiest way to create a random element of `Mat24` is to create a random integer `0 <= x < 244823040`, and to call function `m24num_to_perm(x)`.

3.2.3 Operation of the group `Mat24` on vectors

Elements of `Mat24` operate from the right on vectors `in V = GF(2)**24` or on Golay code or cocode vectors. A function performing such an operation has the name:

```
op_<vector>_<group>
```

where `<vector>` indicates the representation of the vector space and `<group>` indicates the representation of the group. We implement the functions:

```
op_vect_perm, op_gcode_matrix, op_gcode_perm, op_cocode_perm.
```

E.g. function `op_gcode_matrix` operates on a Golay code word (in `gcode` representation) by right multiplying an element `m` of `Mat24` with it. Here element `m` is a 12 times 12 matrix (in `matrix` representation).

3.2.4 Group operation in the group `Mat24`

Multiplication and inversion in the group `Mat24` is supported for the permutation representation `perm`. Therefore we have functions:

```
mul_perm, inv_perm
```

3.2.5 The Parker loop `Pl`

We support the Parker loop `Pl` and also its automorphism group `AutPl`.

An element of `Pl` is a pair `(v, s)`, with `v` a Golay code word and `s` a sign bit, as described in the API reference. We represent the element `(v, s)` as a 13-bit integer, with `v` given by bits 0,. . . ,11 (in `gcode` representation) and the sign `s` given by bit 12. We call this representation of the Parker loop the `ploop` representation. So we can convert and element of `C` in 'gcode' representation to an element of `Pl` in `ploop` representation by adjusting the sign in bit 12.

Function `mul_ploop()` returns the product of two elements of the Parker Loop. Function `inv_ploop()` returns the inverse of ab element of the Parker loop.

Let `theta` be the cocycle for the Parker loop defined in the API reference. For an element `v1` of of `C` or `Pl` in `gcode` or `ploop` representation, the function `ploop_theta(v1)` returns the value `theta(v1)` (which is in `C*`) in `cocode` representation. Function `ploop_cocode(v1, v2)` returns the value of the coycle `theta(v1, v2)`, which is 0 or 1.

3.2.6 The group `AutPl` of standard automorphisms of the Parker loop

An automorphism of the Parker loop is implemented as an array `a` of twelve 32-bit integers. The lowest 13 bits of `a[i]` encode the image of the i-th basis vector of the Parker loop. Here the basis of the Parker loop corresponds to the selected basis of the Golay code, and each basis vector has positive sign.

The bits `13,...,24` of the vectors `a[i]` encode a quadratic form which facilitates computations in `AutPl`, as described in section *Implementing Automorphisms of the Parker loop* in the *Guide for developers*.

This representation of `AutPl` is called the `autpl` representation. We only use the `autpl` representation for elements of `AutPl`.

Function `perm_to_autpl(c, p)` computes an automorphism `m` of the Parker loop created from an element `p` of `Mat24` (given in `perm` representation) and a cocode element `c` (given in `cocode` representation). If `m` is equal to the result of `perm_to_autpl(c, p)`, then we can get back `p` and `c` be computing `p = autpl_to_perm(m)` and `c = autpl_to_cocode(m)`.

Function `cocode_to_autpl(c)` is equivalent to function `perm_to_autpl(c, p0)`, where `p0` is the identity permutation. Note that:

```
perm_to_autpl(c, p) = cocode_to_autpl(c) * perm_to_autpl(0, p).
```

Here `perm_to_autpl(0, p)` is equivalent to *standard representative* of `p` in `AutPl`, and `cocode_to_autpl(c)` is a *diagonal automorphism*, as described in section *Automorphisms of the Parker loop* of the API reference.

Function `op_ploop_autpl(v, m)` applies Parker loop automorphism `m` to element `v` of `Pl` and returns the result.

Function `mul_autpl(m1, m2)` computes the product `m1 * m2` of the Parker loop automorphisms `m1` and `m2`. Function `inv_autpl(m1)` computes the inverse of the Parker loop automorphism `m1`.

3.2.7 Auxiliary functions

Here is an overview of some auxiliary functions in this class. They are described in the corresponding function documentation::

```
bw24              bit weight of the lowest 24 bits of an integer
lsbit24           min(24, least significant bit pos.) for an integer
gcode_weight      weight of a Golay code word in 'gtype' representation
vect_to_bit_list  given a bit vector in V, it computes the lists of
                  the positions of the 0 bits and of the 1 bits of v.
extract_b24       extract bits from bit vector using a 24-bit mask
spread_b24        spread bit vector according to a 24-bit mask
```

3.2.8 Internal operation

For switching from the standard representation to the internal representation we use three tables with `2**8` entries of 24 bit length. For switching back from internal to standard representation we use three other tables of the same format. There are also tables for computing the syndrome of a vector in `V` with respect to the Golay code. There is yet another table for the cocycle `theta` of the Parker loop.

3.2.9 Abbreviations for functions and parameters in this class

The following list of abbreviations used in names of functions allows to infer the action of most functions in this module:

```
Abbreviation  Meaning                                   Data type

assoc         associator (in Golay code or Parker loop)
autpl         automorphism of the Parker loop Pl        uint32_t[12]
bw24          bit weight of the lowest 24 bits of an int
cap           intersection (of Golay code elements)
cocode        element of Golay cocode C*                uint32_t
cocycle       cocycle:  Pl times Pl  ->  {0,1}
comm          commutator (in Golay code or Pl)
gcode         element of Golay code C                   uint32_t
inv           inversion (in Mat24, Pl, or AutPl)
lsbit24       least significant bit of an integer,
              counting bits 0,...,23 only
m24num        number of an element of Mat24             uint32_t
matrix        element of Mat24 as binary matrix
              acting on the Golay code C                uint32_t[12]
mul           multiplication (in Mat24, Pl, or AutPl)
net           Benes network for an element of Mat24     uint32_t[9]
octad         number of an octad, i.e. a Golay code
              element of weight 8                       uint32_t
op            op_<vector>_<operation> means:
              apply <operation> to <vector>
op_all        apply operation to all vectors
perm          element of Mat24 as a permutation         uint8_t[24]
ploop         element of the Parker loop Pl             uint32_t
pow           power operator (in Pl)
scalar_prod   scalar product (of Golay code and cocode)
```

(continues on next page)

(continued from previous page)

```
suboctad      suboctad, see function suboctad_to_cocode
syndrome      syndrome (after decoding Golay code)      uint32_t
theta         cocycle theta: Pl -> C^* in Parker loop
to            <x>_to_<y> means: return representation <y>
              of an object given in representation <x>
vect          vector in V = GF(2)**24                   uint32_t
vintern       vector in V, in internal representation   uint32_t
```

3.2.10 Conventions for parameters in C functions

Parameters of functions are either integers or arrays of integers. Here all integer types are unsigned and of fixed length, such as `uint8_t`, `uint16_t` or `uint32_t`.

The type of a parameter is given by a single letter in the name of the parameter:

```
Name  Meaning                                          Type
a     array specified in documentation of function     unspecified
c     Golay cocode element, represented as 'cocode'    uint32_t
m     permutation in Mat24 or automorphism of Pl
      represented as a bit matrix                      uint32_t[12]
p     permutation in Mat24 represented as 'perm'       uint8_t[24]
u_<x> unsigned integer, e.g.                           unspecified
       u_exp:   integer denoting an exponent
       u_m24:   number of a permutation in Mat24
       u_octad: number of octad, 0 < u_octad < 259
       u_width: integer denoting a bit width
v     vector in V, Golay code C or Parker loop Pl
      represented as vect, vintern, gcode or ploop     uint32_t
```

Integer input parameters have name `u_<x>`, e.g. `u_m24, u_exp`. An integer computed by a function is returned as return value. Input array parameters have a digit as a suffix, e.g.: `v1, v2, m1`. Output array parameters have the suffix `_out`, e.g.: `p_out`. Input/output array parameters have the suffix `_io`, e.g.: `m_io`.

3.2.11 Mapping C functions to python functions

All C functions in module `mat24_functions` are documented. This documentation is not repeated for the corresponding python functions in module `mmgroup.mat24`.

The rules for converting a C function to a python function are as follows:

- To obtain the name of the python function, strip off the prefix `mat24_` from the name of a C functions.
- To obtain the tuple of input parameters for the python function, take the tuple of parameters of the C function and drop are parameters with suffix `_out`.

  In the corresponding python function, an iterable object may be passed where the C function expects a pointer to an integer. The minimum length of that iterable object is either clear from the context or documented in the C function.
- To obtain the return value of the python function, check if the C function has any parameters with suffix `_out`. Then the sequence of returned objects is the sequence of these parameters, possibly preceded by the return value if the C function returns an integer value.

As usual in python, a sequence of length 1 is returned as a single object, and a sequence of length >= 1 is returned as a tuple.

A parameter with suffix `_out` is returned as a list of integers.

- A C function may fail under certain circumstances.

  A failure of a function is indicated in the return value of the function. Details are given in the documentation of the function. The corresponding python function raises ValueError if the C function fails.

- The python function drops the return value of the C function from the sequence of returned python objects in certain cases.

  If the documentation of the C function contains the phrase 'Returns 0 in case of success and (anything else) in case of failure' then the return value is just a status indicator and hence dropped by the python function.

  If the documentation of the C function contains a phrase like 'Returns the length of the list `xxx_out`' then the python function adjusts the length of that returned list appropriately and drops the return value.

- Parameters with suffix `_io` refer to pointers in the C function and hence to iterables in the corresponding python function. Here the sufix `_io` means that the function may modify that iterable object.

3.2.12 C interface

Header files

File `mat24_functions.h` is the header file for `mat24_functions.c`.

Defines

`MAT24_ORDER`

Order of Mathieu group `Mat24`. This is equal to 244823040.

`MAT24_SUBOCTAD_WEIGHTS`

(MAT24_SUBOCTAD_WEIGHTS >> x) & 1 is halved weight of suboctad x (mod 2).

`mat24_def_lsbit24`(v1)

Macro version of function `mat24_lsbit24`.

`mat24_def_lsbit24_pwr2`(v1)

Special macro version of function `mat24_lsbit24`.

This is faster than `mat24_def_lsbit24`, but here `v1` must be power of two.

`mat24_def_parity12`(v1)

Parity of vector `v1` of 12 bits length.

Generate a sequence of statements that replaces `v1` by the bit parity of `v1 & 0xfff`.

`mat24_def_octad_to_gcode`(o)

Eqivalent to *`mat24_def_octad_to_gcode(o)`*

`mat24_def_octad_to_gcode(o)` returns the number of the Golay code word corresponding to octad `o`. Parameter `o` is not checked.

mat24_def_gcode_to_octad(v)

Eqivalent to *mat24_def_gcode_to_octad(v)*

mat24_def_gcode_to_octad(v) returns the number of the octad corresponding to Golay code vector `v`, with `v` in `gcode`. It returns garbage if `v` is not an octad.

mat24_def_not_nonstrict_octad(v)

Check if `v` (or its complement) is an octad.

Returns 0 if `v` (or its complement) is an octad and 1 otherwise.

Vector `v` must be given in `gcode` representation

mat24_def_gcode_to_vect(v)

Convert Golay code element number `v` to a vector in `GF(2)^24`

Macro version of function `mat24_gcode_to_vect`.

mat24_def_syndrome_from_table(t)

Convert entry `t` of table `MAT24_SYNDROME_TABLE` to syndrome.

An entry `t` of the table `MAT24_SYNDROME_TABLE` encodes an odd cocode syndrome. The macro returns that syndrome as a bit vector.

mat24_def_suboctad_weight(u_sub)

Equivalent to mat24_suboctad_weight(u_sub)

Enums

enum **mat24_rand_flags**

Flags describing subgroups of the Mathieu group M_{24}.

This enumeration contains flags describing some subgroups of the Mathieu group M_{24} fixing certain subsets (or sets of subsets) of the set $\tilde{\Omega} = \{0, \ldots, 23\}$ on which the group M_{24} acts. Intersetions of these subgroups may be described by combining these flags with the bitwise or operator `|`. For each flag we state the set being fixed.

Values:

enumerator **MAT24_RAND_2**

fixes $\{2, 3\}$

enumerator **MAT24_RAND_o**

fixes $\{0, \ldots, 7\}$

enumerator **MAT24_RAND_t**

fixes $\{\{8i, \ldots, 8i + 7\} \mid i < 3\}$

enumerator **MAT24_RAND_s**

fixes $\{\{4i, \ldots, 4i + 3\} \mid i < 6\}$

enumerator **MAT24_RAND_l**

fixes $\{\{2i, 2i + 1\} \mid 4 \leq i < 12\}$

enumerator **MAT24_RAND_3**

fixes $\{1, 2, 3\}$

Functions

static inline uint32_t **mat24_inline_cocode_to_suboctad**(uint32_t c1, uint32_t v1, uint32_t u_strict)

Inline version of function `mat24_cocode_to_suboctad`

static inline uint32_t **mat24_inline_suboctad_to_cocode**(uint32_t u_sub, uint32_t u_octad)

Inline version of function `suboctad_to_cocode`

int32_t **mat24_check_endianess**(void)

Check endianess of the machine.

Return 0 if machine is little endian, 1 if it is Big endian, and -1 if endianess could not be detected.

C functions for the Mathieu group M_{24}

File `mat24_tables.c` contains tables that are used in the file `mat24_functions.c`.

Variables

const uint16_t **MAT24_OCT_DEC_TABLE**[759]

Table for converting `octad` to `gcode` representation.

The public macro `mat24_def_octad_to_gcode` uses this table

const uint16_t **MAT24_OCT_ENC_TABLE**[2048]

Table for converting `gcode` to `octad` representation.

The public macro `mat24_def_gcode_to_octad` uses this table

const uint16_t **MAT24_THETA_TABLE**[]

Table containing data about the Golay code.

For `0 <= d < 0x800` entry d contains the following information the code word `d`, with `d` in `gcode` representation.

Bit 11,. . . ,0: `mat24_ploop_theta(d)`

Bit 14,. . . ,12: Bit weight of code word `d` in `GF(2)**24` divided by 4

Bit 15: reserved

We have `d**2 = (-1)**<Bit 12 of entry d>` for `d` in the Parker loop.

const uint8_t **MAT24_OCTAD_ELEMENT_TABLE**[759 * 8]

For `0 <= i < 759`, the entries `8*i,...8*i+7` in the table `MAT24_OCTAD_ELEMENT_TABLE` are the bit positions of the octad with the number `i`.

File `mat24_functions.c` contains the C implementation of the functionality of Python module `mmgroup.mat24`

This covers the Golay code, its cocode, the Parker loop, the Mathieu group Mat24, and the group of standard automorphisms of the Parker loop.

Functions

uint32_t **`mat24_lsbit24`**(uint32_t v1)

Return position of least significant bit of an integer.

The function returns the minimum of the number 24 and the position of the least significant bit of `v1`. It uses a De Bruijn sequence.

uint32_t **`mat24_bw24`**(uint32_t v1)

Returns the bit weight of the lowest 24 bits of `v1`.

uint32_t **`mat24_vect_to_bit_list`**(uint32_t v1, uint8_t *a_out)

Stores the positions of 1-bits of a bit vector to an array.

Let `w` be the bit weight of the bit vector `v1 & 0xffffff`, i.e. number of bits of `v1` at positions `< 24` equal to one. Then the ordered bit positions where the corresponding bit of `v1` is 1 are stored in `a_out[0],...,a_out[w-1]`.

Then `(v1 & 0xffffff)` has `24 - w` zero bits. The ordered list of the positions of these zero bits is stored in `a_out[w],...,a_out[23]`.

The function returns the bit weight `w`.

uint32_t **`mat24_vect_to_list`**(uint32_t v1, uint32_t u_len, uint8_t *a_out)

Stores the positions of 1-bits of a bit vector to an array.

Let `w` be the minimum of the input `u_len` and the bit weight of the bit vector `v1 & 0xffffff`, i.e. number of bits of `v1` at positions `< 24` equal to one. Then the first `w` bit positions where the corresponding bit of `v1` is 1 are stored in `a_out[0],...,a_out[w-1]` in natural order. The function returns `w`.

For small values `w` this function is faster than function `mat24_vect_to_bit_list`.

uint32_t **`mat24_extract_b24`**(uint32_t v1, uint32_t u_mask)

Extract the bits of 24-bit vector `v1` given by the mask `u_mask`

If `u_mask` has bits equal to one at positions `i_0, i_1, ..., i_k` (in ascending order) then the bit of `v1` at position `i_j` is copied to the bit at position `j` of the return value for `j = 0,...,k`.

uint32_t **`mat24_spread_b24`**(uint32_t v1, uint32_t u_mask)

Spread bits of 24-bit vector `v1` according to the mask `u_mask`

If `u_mask` has bits equal to one at positions `i_0, i_1, ..., i_k` (in ascending order) then the bit of `v1` at position `j` is copied to the bit at position `i_j` of the return value for `j = 0,...,k`.

uint32_t **`mat24_vect_to_vintern`**(uint32_t v1)

Convert bit vector `v1` in `GF(2)^24` from `vector` to `vintern` representation.

uint32_t **`mat24_vintern_to_vect`**(uint32_t v1)

Convert bit vector `v1` in `GF(2)^24` from `vintern` to `vector` representation.

uint32_t **`mat24_vect_to_cocode`**(uint32_t v1)

Return Golay cocode element corresponding to a bit vector in `GF(2)^24`.

This amounts to reducing the vector `v1` (given in `vector` representation) modulo the Golay code. The function returns the cocode element corresponding to `v1` in `cocode` representation.

uint32_t **`mat24_gcode_to_vect`**(uint32_t v1)

Convert Golay code element number `v1` to a vector in `GF(2)^24`

Input `v1` is a Golay code element in `gcode` representation. The function returns the bit vector corresponding to `v1` in `vector` representation.

uint32_t **`mat24_cocode_to_vect`**(uint32_t c1)

Return a vector in `GF(2)^24` corresponding to cocode element.

Here `c1` is the number of a cocode element in `cocode` representation. One of `2**12` possible preimages of `c1` in `GF(2)^24` is returned in `vector` representation.

uint32_t **`mat24_vect_to_gcode`**(uint32_t v1)

Return a vector in `GF(2)^24` as a Golay code element.

If the vector `v1` (given in `vector` representation) is in the Golay code then the function returns the number of that Golay code word. Thus the return value is in `gcode` representation.

If `v1` is not in the Golay code then the function returns `(uint32_t)(-1)`.

uint32_t **`mat24_gcode_to_octad`**(uint32_t v1, uint32_t u_strict)

Return a Golay code vector as an octad.

If `u_strict` is even then the function acts as follows:

If the Golay code vector `v1` (given in `gcode` representation) is an octad or a complement of an octad then the function returns the number of that octad. Thus the return value is in `octad` representation. Then we have `0 <= octad(v1, strict) < 759`.

If `v1` is not a (possibly complemented) octad then the function returns `(uint32_t)(-1)`.

If `u_strict` is odd then the function returns `(uint32_t)(-1)` also in case of a complemented octad `v1`.

uint32_t **`mat24_vect_to_octad`**(uint32_t v1, uint32_t u_strict)

Return a vector in `GF(2)^24` as an octad.

If `u_strict` is even then the function acts as follows:

If the vector `v1` (given in `vector` representation) is an octad or a complement of an octad then the function returns the number of that octad. Thus the return value is in `octad` representation. Then we have `0 <= octad(v1, strict) < 759`.

If `v1` is not a (possibly complemented) octad then the function returns `(uint32_t)(-1)`.

If `u_strict` is odd then the function returns `(uint32_t)(-1)` also in case of a complemented octad `v1`.

uint32_t **`mat24_octad_to_gcode`**(uint32_t u_octad)

Convert an octad to a Golay code vector.

Given an octad `u_octad` (in `octad` representation), the function returns the number of the corresponding Golay code number in `gcode` representation.

There are 759 octads. The function returns `(uint32_t)(-1)` in case `u_octad >= 759`.

uint32_t **`mat24_octad_to_vect`**(uint32_t u_octad)

Convert an octad to a bit vector in `GF(2)^24`.

Given an octad `u_octad` (in `octad` representation), the function returns bit vector corresponding to that octad in `vector` representation.

There are 759 octads. The function returns `(uint32_t)(-1)` in case `u_octad >= 759`.

uint32_t **`mat24_cocode_syndrome`**(uint32_t c1, uint32_t u_tetrad)

Return Golay code syndrome of cocode element `c1`.

Here `c1` is a cocode element in `cocode` representation. mat24_cocode_syndrome(c1, u_tetrad) is equivalent to mat24_syndrome(mat24_cocode_to_vect(c1), u_tetrad).

The function returns a Golay code syndrome as described in the documentation of function *mat24_syndrome()*.

uint32_t **mat24_syndrome**(uint32_t v1, uint32_t u_tetrad)

Return Golay code syndrome of word `v1`.

Here `v1` is an arbitrary word in `GF(2)**24` in `vector` representation. The function returns a Golay code syndrome of `v1` (of minimum possible bit weight) as a bit vector in `vector` representation.

Such a syndrome is unique if it has weight less than 4. In that case the unique syndrome is returned.

If the minimum weight of the syndrome is four then the six possible syndroms form a partition of the the underlying set of 24 elements. In this case we return the syndrome of bit weight four where the bit at position `u_tetrad` is set. Therefore parameter `u_tetrad` must satisfy `0 <= u_tetrad <= 24`; otherwise the function fails.

If the minimum weight of the sydrome is at most three, parameter `u_tetrad` must satisfy `0 <= u_tetrad < 24`; otherwise the function fails.

The function returns `(uint32_t)(-1)` in case of failure.

uint32_t **mat24_gcode_weight**(uint32_t v1)

Returns bit weight of Golay code word `v1` divided by 4.

Here `0 <= v1 < 4096` is the number of a Golay code word, i.e. `v1` is given in `gcode` representation.

uint32_t **mat24_gcode_to_bit_list**(uint32_t v1, uint8_t *a_out)

Store bit positions of Golay code `v1` in array `a_out`

Here `0 <= v1 < 4096` is the number of a Golay code word, i.e. `v1` is given in `gcode` representation. The Golay code word `v1` is stored in the array referred by `a_out` as an ordered list of the positions of the bits being set in the word `v1`.

That array must have physical length at least 24. The function returns the actual length of the returned array, which is equal to the bit weight of the word `v1`.

uint32_t **mat24_cocode_weight**(uint32_t c1)

Return the minimum possible weight of the cocode vector `c1`

Here `c1` is a cocode element in `cocode` representation.

uint32_t **mat24_cocode_to_bit_list**(uint32_t c1, uint32_t u_tetrad, uint8_t *a_out)

Store Golay code syndrome of cocode word `c1` in an array.

Here `c1` is an cocode word in cocode representation. The function stores the sorted bit positions of the syndrome of `c1` in the array referred by `a_out` and returns the actual length of that array, which is the weight of the syndrome. The array referred by `a_out` must have physical length at least 4.

Such a syndrome is unique if it has weight less than 4. In that case the unique syndrome is returned.

If the minimum weight of the syndrome is four then the six possible syndroms form a partition of the the underlying set of 24 elements. In this case we return the syndrome of bit weight four where the bit at position `u_tetrad` is set. Therefore parameter `u_tetrad` must satisfy `0 <= u_tetrad < 24`; otherwise the function fails.

If the minimum weight of the sydrome is at most three, parameter `u_tetrad` must satisfy `0 <= u_tetrad <= 24`; otherwise the function fails.

The function returns `(uint32_t)(-1)` in case of failure.

uint32_t **mat24_cocode_to_sextet**(uint32_t c1, uint8_t *a_out)

Store a cocode word `c1` in array `a_out` as a sextet.

Here `c1` is an cocode word in cocode representation. That cocode word must correspond to a syndrome of length four, i.e. the syndrome must be a tetrad. Otherwise the function fails.

The function stores the six tetrads that make up the sextet `c1` in `a_out[4*i],...,a_out[4*i+3]` for `i = 0,...,5`. The (ordered) tetrads are stored in lexical order.

The function returns `(uint32_t)(-1)` if `c1` has not minimum weight 4.

uint32_t **`mat24_scalar_prod`**(uint32_t v1, uint32_t c1)

Return scalar product of Golay code and cocode vector.

`v1` is a Golay code vector in 'gcode' representation, `c1` is a cocode vector in cocode representation.

Actually the function returns the bit parity of `v1 & c1 & 0xfff`.

uint32_t **`mat24_cocode_to_suboctad`**(uint32_t c1, uint32_t v1, uint32_t u_strict)

Convert cocode element `c1` to suboctad of octad `v1`

The function converts a cocode element `c1` (in `cocode` representation) and an octad `v1` (in `gcode` representation) to a suboctad.

The function returns `(o << 6) + u_sub`. Here `o` is the octad number corresponding to octad `v1` and `u_sub` is the suboctad number corresponding to the cocode element `c1`, if `u_octad` is an octad and `c1` is an even subset of `u_octad`.

Each octad `v1` has 64 even subsets, when each subset of `v1` is identified with its complement in `v1`. These subsets are called suboctads. Let `b_0, ..., b_7` be the elements of the octad `v1` in the order as returned by applying function `mat24_octad_entries` to octad number `o`. Then the even subset `(b_0 , b_i)` has suboctad number `2**(i-1)` for `i = 1,...,6`. Combining suboctads by symmetric difference corresponds to combining their numbers by `xor`. The empty subocatad has number zero. This yields a one-to-one correspondence between the integers `0,...,63` and the suboctads of a fixed octad `v1`, when identifying a suboctad ith its complement.

At present elements of the octads are ordered in natural order. But this is subject to change!

The function fails if `v1` is not a octad or `c1` cannot be represented as an even subset of `v1`. If `v1` is a complement of an octad `o` the `o` is taken instead of `v1`. If `u_strict` is set then the pair `(v1, c1)` must correspond a short Leech lattice vector. Otherwise is suffices to specify `v1` up to an additive term Ω.

The function returns `(uint32_t)(-1)` in case of failure.

uint32_t **`mat24_suboctad_to_cocode`**(uint32_t u_sub, uint32_t u_octad)

Convert even suboctad of octad to cocode representation.

The function converts a suboctad `u_sub` (in `suboctad` representation) of an octad `u_octad` (in `octad` representation) to a cocode element. It returns that cocode element in `cocode` representation. This is a partial inverse of function *`mat24_suboctad_to_cocode()`*. The ordering of the suboctads is described in that function.

The function fails if `u_octad` does not represent an octad. It returns `(uint32_t)(-1)` in case of failure.

uint32_t **`mat24_octad_entries`**(uint32_t u_octad, uint8_t *a_out)

List the entries of an octad.

The function writes the list of entries of octad `u_octad` (in `octad` representation) into the array `a_out` of length 8. The order of these entries is the order used for the conversion of suboctads. It may differ from the natrual order.

The function returns 0 if `u_octad` is the number of an octad and `(uint32_t)(-1)` otherwise.

uint32_t **`mat24_suboctad_weight`**(uint32_t u_sub)

Return parity of halved bit weight of the even suboctad.

Here parameter `u_sub` is the number of a suboctad. A suboctad cooresponds to a subset of an octad of even parity. The function returns 0 is the bit weight of that subset is divisible by four and 1 otherwise.

The numbering of suboctads is described in the documentation of function *mat24_suboctad_to_cocode()*.

uint32_t **mat24_suboctad_scalar_prod**(uint32_t u_sub1, uint32_t u_sub2)

Return scalar product of two suboctads.

The function returns the scalar product of the two suboctads with the numbers `u_sub1, u_sub2`.

Here the scalar product is the parity of the vector `u_sub1 & u_sub2` when `u_sub1` and `u_sub2` are given as subsets of an octad in vector notation.

But in this functions parameters `u_sub1, u_sub2` are suboctad numbers as documented in function *mat24_suboctad_to_cocode()*.

uint32_t **mat24_cocode_as_subdodecad**(uint32_t c1, uint32_t v1, uint32_t u_single)

Represent a cocode element as a subset of a docecad.

Given a Golay cocode element `c1` (in `cocode` representation) and a dodecad `v1` (in `gcode` representation), the function returns a bit vector `c_out` equivalent to the cocode word `c1`, which is a subset of the dodecad `d1`. This is possible if the scalar product of `c1` and the complement of `v1` is even. Otherwise the function fails.

The user may specify a bit position `0 <= u_single < 24` disjoint from the bits of dodecad `d1`. Then that bit of the return value `c_out` will be set if the scalar product mentioned above is odd, and the function succeeds also in this case.

The intersection of `c_out` with `v1` has bit weight at most 6. If that bit weight is equal to 6 then `c_out` contains the least significant bit of the bit vector corresponding to `v1`.

The function fails if `v1` is not a dodecad. It returns `(uint32_t)(-1)` in case of failure.

uint32_t **mat24_ploop_theta**(uint32_t v1)

Returns the theta function for the Parker loop.

Here function `theta()` is a quadratic function from the Golay code `C` to the cocode `C*`. Parameter `v1` of function `theta` is a Golay code word in `gcode` representation. The result of the `theta` function is returned as a Golay cocode word in `cocode` representation.

The cocycle of the Parker loop is given by:

```
    cocycle(v1, v2) =   mat24_scalar_prod(theta(v1), v2),
```

where *mat24_scalar_prod()* computes the scalar product.

The function evluates the lower 12 bits of `v1` only. Thus `v1` may also be an element of the Parker loop.

uint32_t **mat24_ploop_cocycle**(uint32_t v1, uint32_t v2)

Returns the cocycle of the Parker loop.

Here parameters `v1` and `v2` are Golay code vectors in `gcode` representations or elements of the Parker loop, coded as in function `mat24_mul_ploop`. Then the Parker loop product of `v1` and `v2` is given by

```
v1 (*) v2  =  v1 ^ v2 * (-1)**cocycle(v1, v2).
```

uint32_t **mat24_mul_ploop**(uint32_t v1, uint32_t v2)

Returns the product of two elements of the Parker loop.

Here the Parker loop elements `v1` and `v2` are integers coded as follows:

```
bit 0,...,11:   a Golay code word in ``gcode`` representation

bit 12:         Parker loop sign
```

The other bis of`v1` and `v2` are ignored.

uint32_t **mat24_pow_ploop**(uint32_t v1, uint32_t u_exp)

Returns a power of an element of the Parker loop.

Here `v1` is a the Parker loop element coded as in function *mat24_mul_ploop()*. `u_exp` is the exponent. The function returns the power `v1 ** exp` as an element of the Parker loop.

E.g. mat24_pow_ploop(v1, 3) is the inverse of `v1`.

uint32_t **mat24_ploop_comm**(uint32_t v1, uint32_t v2)

Return commutator of Golay code words `v1` and `v2`

This is equal to 0 if the intersection of the bit vectors `v1` and `v2` has bit weight 0 mod 4, and equal to 1 is that intersection has bit weight 2 mod 4. Words `v1` and `v2` must be given in `gcode` representation.

For Parker loop elements `v1` and `v2` (coded as in function `mat24_mul_ploop`) the commutator of `v1` and `v2` is equal to

```
  (-1) ** mat24_ploop_comm(v1, v2),
```

where ** denotes exponentiation.

uint32_t **mat24_ploop_cap**(uint32_t v1, uint32_t v2)

Return intersection of two Golay code words as cocode word.

Here `v1` and `v2` are Golay code words in `gcode` representation. The result is a cocode word returned in `cocode` representation.

uint32_t **mat24_ploop_assoc**(uint32_t v1, uint32_t v2, uint32_t v3)

Return associator of Golay code words `v1, v2,` and `v3`

This is the parity of the intersection of the bit vectors `v1, v2,` and `v3`. So the function returns 0 or 1. Vectors `v1, v2, v3` are in `gcode` representation.

The associator of three Parker loop elements `v1, v2, v3` is equal to

```
(-1) ** mat24_ploop_assoc(v1, v2, v3) .
```

Here `v1, v2, v3` are encoded as in function *mat24_mul_ploop()*.

uint32_t **mat24_ploop_solve**(uint32_t *p_io, uint32_t u_len)

Return cocode element that kills signs of Parker loop elements.

Here `p_io` refers to an array of `u_len` Parker loop elements are coded as in function *mat24_mul_ploop()*. The function tries to find a cocode element that makes all these Parker loop elements positive, when operating on them as a diagonal automorphism. The function returns the least cocode element in lexical order satisfying that condition in the bits `0,...,11` of the return value. For that order we assume that lower bits have higher valence. If no such cocode element exists, the function fails.

We set bit 12 of the return value to indicate a failure.

The array `p_io` is destroyed. More specifically, the first `k` entries of that array are changed to an array of linear independent Parker loop elements. When these `k` elements are mapped to positive Parker loop elements, this also yields a solution of the original problem. If the problem cannot be solved then we put `p_io[k-1] = 0x1000`.

The function returns the value `k` in bits `31,...,16` of the result.

uint32_t **mat24_perm_complete_heptad**(uint8_t *p_io)

Complete permutation in the Mathieu group `Mat24` from 7 images.

This is an auxilary function for function *mat24_perm_from_heptads()*. We use the terminology introduced in that function.

The function completes the array `p_io` to a permutation `p` in the Mathieu group `Mat24`. On output, permutation `p` is given as a mapping `i -> p_io[i]` for `i = 0,...,23`.

On input, the images `p_io[i]` must be given for `i = 0,1,2,3,4,5,8`; the other entries of `p_io` are ignored.

The set `(p_io[i], i = 0,1,2,3,4,5,8)` must be an umbral heptad with distiguished element `p_io[8]`. Then the mapping `i -> p_io[i], i = 0,1,2,3,4,5,8` is a feasible mapping between umbral heptads; it extends to a unique permutation in `Mat24`. Note that `8` is the distingished element of the umbral heptad `(0,1,2,3,4,5,8)`.

The function returns 0 if the mapping given on input can be extended to an element of `Mat24`, and a nonzero value otherwise.

Implementation idea:

We choose pentads, i.e. subsets of size 5 of the set `(0,....,23)` that consist of known values `p_io[i]`. We calculate the syndromes of such pentads, which are triads, i.e. sets of size three. Calculating the syndromes of the preimages of these pentads we obtain mappings between triads. Intersecting triads in a suitable way we obtain mappings between singletons, and hence peviously unknown images of elements of the set `(0,....,23)`.

uint32_t **`mat24_perm_check`**(uint8_t *p1)

Check if permutation is in in the Mathieu group `Mat24`.

The function checks the mapping `i -> p1[i]`, `i = 0,...,23`.

It returns zero if that mapping is a permutation in `Mat24` and a nonzero value otherwise.

The implementation uses function *mat24_perm_complete_heptad()*.

uint32_t **`mat24_perm_complete_octad`**(uint8_t *p_io)

Complete an octad given by 6 elements of it.

Given entries `p_io[i], i = 0,1,2,3,4,5`, we calculate values `p_io[6], p_io[7]` such that the set `(p_io[i], 0 <= i < 8)` is an octad. Furthermore, we order the values `p_io[6], p_io[7]` in such way that the mapping `i -> p_io[i]` may be extended to a permutation in the grpup `Mat24`. This restrtiction determines the order uniquely. Note that the set `0,...,7` is an octad, which is called the standard octad.

The set `p_io[i], i = 0,1,2,3,4,5` must be a subset of an octad; otherwise the function fails. The function returns 0 in case of success and `(uint32_t)(-1)` in case of failure.

The implementation is a simplified version of function *mat24_perm_complete_heptad()*.

uint32_t **`mat24_perm_from_heptads`**(uint8_t *h1, uint8_t *h2, uint8_t *p_out)

Complete a mapping to a permutation in the Mathieu group `Mat24`

A permutation in the Mathieu group `Mat24` is a mapping from the set `(0,...,23)` to itself. The function completes the mapping `h1[i] -> h2[i]`, `0 <= i < 7`, `0 <= h1[i], h2[i] < 24` to a permutation `i -> p_out[i]`, `0 <= i < 24` in the group `Mat24`. The result is returned in the array `p_out[i]`.

The sets `h1[i], 0 <= i < 7` and `h2[i], 0 <= i < 7` must be umbral heptads, and the mapping from `h1` to `h2` must be feasible.

An umbral heptad is a set of seven elements of the set `(0,...,23)` which is not a subset of an octad. The syndrome of an umbral heptad, i.e. the smallest set equivalent to the heptad modulo the Golay code, is a singleton containing exactly one element of the umbral heptad. That element is called the distiguished element of the heptad. A feasible mapping from an umbral heptad to another umbral heptad is a mapping that maps the distiguished element of the first heptad to the distiguished element of the second heptad.

It can be shown that a feasible mapping from an umbral heptad to another umbral heptad extends to a unique element of the Mathieu group.

The function returns 0 if the mapping `h1[i] -> h2[i]` can be extended to an element of `Mat24` and `(uint32_t)(-1)` otherwise.

The implementation uses function *mat24_perm_complete_heptad()*.

uint32_t **`mat24_perm_from_map`**(uint8_t *h1, uint8_t *h2, uint32_t n, uint8_t *p_out)

Complete a mapping to a permutation in the Mathieu group `Mat24`

A permutation in the Mathieu group `Mat24` is a mapping from the set `(0,...,23)` to itself. The function tries to complete the mapping `h1[i] -> h2[i]`, `0 <= i < n`, `0 <= h1[i], h2[i] < 24` to a permutation `i -> p_out[i]`, `0 <= i < 24` in the Mathieu group `Mat24`. In case of success, such a permutation is stored in the array `p_out`.

The function returns

-1 if the mapping `h1[i] -> h2[i]` does not extend to a legal permutation of the numbers 0,…,23. Note that duplicate entries in `h1` or `h2` are illegal.

0 if no such permutation exists in the Mathieu group `Mat24`.

1 if the mapping `h1[i] -> h2[i]` extends to a unique permutation in `Mat24`.

2 if if the mapping `h1[i] -> h2[i]` can be completed to several permutations in `Mat24`, and not all entries `h1[i]` can be covered by an octad. This may happen in case `n = 6` only.

3 if if the mapping `h1[i] -> h2[i]` can be completed to several permutations in `Mat24`, and all entries `h1[i]` can be covered by an octad.

If the return value is greater then zero then a suitable permutation in `Mat24` is returned in the array referred by `p_out`. The function computes the lowest permutation (in lexical order) that maps `h1` to `` `h2 ``.

Caution:

Some input mappings allow several output permutations. Changing the specification of this function such that the same input leads to a different output permutation destroys the interoperability between different versions of the project!!

uint32_t **`mat24_m24num_to_perm`**(uint32_t u_m24, uint8_t *p_out)

Compute permutation in the Mathieu group `Mat24` from its number.

The Mathieu group has order `244823040`. We assign numbers `0 <= n < 244823040` to the elements of `Mat24`, in lexicographic order, with `0` the number of the neutral element. This is just a convenient way to refer to an element of `Mat24`.

The function calculates the permutation with the number `u_m24` and stores it in the array `p_out` as a mapping `i -> p_out[i]`. `0 <= u_m24 < 244823040` must hold; otherwise the function fails.

The function returns 0 in case of success and `(uint32_t)(-1)` in case of failure.

uint32_t **`mat24_perm_to_m24num`**(uint8_t *p1)

Compute number of a permutation in the Mathieu group `Mat24`

This is the inverse of function *mat24_m24num_to_perm()*.

Given a permutation `i -> p1[i], 0 <= i < 24`, the function returns the number `n` of that permutation. We have `0 <= n < 244823040`, as described in function `mat24_m24num_to_perm`.

The function returns garbage if `p1` is not a valid permutation in `Mat24`. One may use function *mat24_perm_check()* to check if permutation `p1` is in `Mat24`.

void **`mat24_perm_to_matrix`**(uint8_t *p1, uint32_t *m_out)

Convert permutation in the Mathieu group `Mat24` to bit matrix.

The input of the function is the permutation `p:  i -> p1[i]`, `0 <= i < 24` which must be in the Mathieu group `Mat24`.

The function computes a `12 times 12` bit matrix `m`, acting on a Golay code vector `v` (in `gcode` representation) by right multiplication. Then we have `v * m = p(v)`. Bit `m[i,j]` is stored in bit `j` of the the integer `m_out[i]`.

Output `m_out[i], 0 <= i < 12` contains garbage if `p` is not in `Mat24`.

Implementation idea:

In the standard basis of `GF(2)**24`, that operation corresponds to a permutation. We have precomputed a matrix converting that standard basis to an internal basis, where Golay code words are visible, and also the inverse of that matrix. Thus the operation is just an (optimized) sequence of matrix multiplications.

void **mat24_matrix_to_perm**(uint32_t *m1, uint8_t *p_out)

Convert bit matrix to permutation in the Mathieu group `Mat24`

This is the inverse of function *mat24_perm_to_matrix()*

The input `m1` of the function is a bit matrix that maps a Golay code word `v` to `p(v) = v * m1`, as described in function *mat24_perm_to_matrix()*.

The function computes the permutation `p:  i -> p_out[i]`, `0 <= i < 24`, from that matrix and stores the result in the output vector `p_out` of length 24.

Output `p_out[i], 0 <= i < 24` contains garbage if `m1` is not bit matrix corresponding to an element of `mat24`.

Implementation idea:

We could have reversed the operation of function *mat24_perm_to_matrix()*, but the following implememtation is faster:

Converting rows of matrix `m1` from `gcode` to `standard` representation yields the images of some Golay code words as bit vectors. Intersecting these bit vectors in a suitable way yields the images of singletons and hence the requestend permutation.

void **mat24_matrix_from_mod_omega**(uint32_t *m1)

Complete bit matrix for Mathieu group `Mat24` from submatrix.

Let the input `m1` of the function be a 12 times 12 bit matrix that maps a Golay code word `v` to `p(v) = v * m1`, as described in function `mat24_perm_to_matrix()`.

There are cases where the first 11 rows and columns of `m1` can be deduced from an external source, but the last row and column is unknown. This means the operation of an element of `Mat24` on the Golay code is know modulo the code word `Omega = (1,...,1)` only. This function completes such an 11 times 11 bit matrix to a 12 times 12 matrix in place.

static uint32_t **dodecad_to_heptad**(uint8_t *d1, uint8_t *h_out)

Compute a (unique) heptad from a dodecad.

This is a (rather technical) auxiliary function for function `mat24_perm_from_dodecads`

Let `d1` be a dodecad as in function *mat24_perm_from_dodecads()*.

There is a unique umbral heptad `h`, as defined in the documentation of function *mat24_perm_from_heptads()*, satisfying the properties described below.

The function evaluates the first 9 elements `d1[i], 0 <= i < 9`. It fails if these elements are not a subset of a dodecad or not pairwise disjoint.

The function returns 0 in case of success and `(uint32_t)(-1)` in case of failure.

Properties of heptad `h`:

`h` contains `d1[i]` for `0 <= i < 5` and also the unique element `h_5` in the intersection of `d1` and the syndrome `S5` of the set `(d1[i], 0 <= i < 5)`.

Let `S5 = (h_5, h_6, h_7)` such that `Mat24` contains a mapping that maps `i` to `d1[i]` for `i < 5` and `i` to `h_i` for `i = 5, 6, 7`. This determines `h_6` uniquely. Then `h_6` is not in the dodecad `d1`. Let `T` be the tetrad containing the set `(d1[0], d1[1], d1[2], h_6)`. Tetrad `T` contains exactly one set `U` intersecting `d1` in 3 elements disjoint to `(d1[i], 0 <= i < 5)`. Then heptad `h` contains the element `d_7` of the singleton `U \ d1` as ts distinguished element.

The function puts `h_out[i] = d1[i]` for `0 <= i < 5` and `h_out[5] = h_5, h_out[6] = d_7`.

uint32_t **mat24_perm_from_dodecads**(uint8_t *d1, uint8_t *d2, uint8_t *p_out)

Find permutation in `Mat24` mapping one dodecad to another.

A dodecad is a word of the Golay code of weight 12. Given two dodecads `d1, d2`, and five elements `d1[0], ...,d1[4]` of `d1`, and also five elements `d2[0],...,d2[4]` of `d2`, there is a unique permutation `p` in `Mat24` with the following properties:

```
d1     is mapped to   d2   ,

d1[i]  is mapped to   d2[i]   for  0 <= i < 5 .
```

On input the function takes two dodecads `d1, d2` as arrays of integers, and it computes a permutation `p: i -> p_out[i]` satisfying the properies given above. In case of success it stores `p` in the array `p_out`. Only the first 9 elements `d1[i], d2[i], 0 <= i < 9` of `d1` and `d2` are evalutated. There is at most one dodecad containing `d1[i], 0 <= i < 9` and at most one dodecad containing `d2[i], 0 <= i < 9`, assuming `d1[i] != d1[j]` and `d2[i] != d2[j]` for `i != j`.

The function fails if the evaluated entries of `d1` or of `d2` are not a subset of a dodecad or not pairwise disjoint.

The function returns 0 in case of success and `(uint32_t)(-1)` in case of failure.

The implementation calls function dodecad_to_heptad() for contructing (unique) heptads `h1` and `h2` from `d1` and `d2`. Then it calls function *mat24_perm_from_heptads()* for computing the unique permutation in `Mat24` that maps `h1` to `h2`.

uint32_t **mat24_op_vect_perm**(uint32_t v1, uint8_t *p1)

Apply a permutation in the Mathieu group `Mat24` to a bit vector.

Apply the permutation `p: i -> p1[i]` to the bit vector `v1`. This maps bit `i` of `v1` to bit `p1[i]` of the returned result.

uint32_t **mat24_op_gcode_matrix**(uint32_t v1, uint32_t *m1)

Apply a `12 times 12` bit matrix to a Golay code vector.

A matrix `12 times 12` bit matrix `m` must be encoded in the input parameter `m1` as specified in function *mat24_perm_to_matrix()*. The function returns the matrix product `v1 * m` as bit vector. Input `v1` and the return value are Golay code words given in `gcode` representation.

uint32_t **mat24_op_gcode_perm**(uint32_t v1, uint8_t *p1)

Apply a permutation in the group `Mat24` to a Golay code vector.

Apply the permutation `p: i -> p1[i]` (which must be an element of the Mathieu group Mat24) to the Golay code word `v1`, with `v1` given in `gcode` representation.

The function returnes the permuted Golay code word in `gcode` representation.

uint32_t **mat24_op_cocode_perm**(uint32_t c1, uint8_t *p1)

Apply a permutation in the group `Mat24` to a Golay cocode element.

Apply the permutation `p: i -> p1[i]` (which must be an element of the Mathieu group Mat24) to the Golay cocode element `c1`, with `c1` given in `cocode` representation.

The function returnes the permuted cocode word in `cocode` representation.

void **mat24_mul_perm**(uint8_t *p1, uint8_t *p2, uint8_t *p_out)

Compute product of two permutations in the Mathieu group `Mat24`

Here inputs `p1, p2` must be permutations represented as mappings `i -> p1[i], i -> p2[i]`. The function computes the product `p1 * p2` and stores it in the array `p_out` in the same form. Thus `p_out[i] = p2[p1[i]]`.

Input errors are not detected, but output buffer overflow is prevented. Any overlap between `p1, p2,` and `p_out` is possible.

void **mat24_inv_perm**(uint8_t *p1, uint8_t *p_out)

Compute the inverse of a permutation in the Mathieu group `Mat24`

Here input `p1` must be a permutation represented as mapping `i -> p1[i]`. The function computes the inverse of `p1` and stores it in the array `p_out` in the same form. Thus `p_out[p1[i]] = i`.

Input errors are not detected, but output buffer overflow is prevented. Any overlap between `p1` and `p_out` is possible.

void **mat24_autpl_set_qform**(uint32_t *m_io)

Auxiliary function for function *mat24_perm_to_autpl()*

Given a Parker loop autmorphism `a`, the function computes a quadratic form `qf` on the Golay code defined as follows:

`qf(g[i]) = 0` for all basis vectors `g[i]` of the standard basis of the Golay code. Furthermore we have

```
qf(v1 + v2) = qf(v1) + qf(v2) + b(v1, v2) ,
```

where `b` is a bilinear form on the Golay code defined by

```
b(x,y) = theta(p(x), p(y)) + theta(x, y)    (mod 2).
```

Here `p` is the element of the group `Mat24` obtained by taking the automorphism `a` modulo sign, and `theta` is the cocycle of the Parker loop. Then `b` is an alternating bilinear form by [Seys20], Lemma 4.1.

Let `b[i,j] = b(g[i], g[j])` where `g[i]` is the `i-th` basis vector of the Golay code in our selected standard basis.

Input `m_io` represents the Parker loop automorphism `a` as documented in function *mat24_perm_to_autpl()*. Here we modify the array `m_io` by storing the bit `b[i,j]` in bit 13+j of entry `m_io[i]` for `i > j`.

So the quadratic form `qf` is now also stored in the array `m_io` representing the automorphsm `a`. As explained in [Seys20] this facilitates computation in the automorphism group of the Parker loop.

void **mat24_perm_to_autpl**(uint32_t c1, uint8_t *p1, uint32_t *m_out)

Construct a Parker loop automorphism.

The function combines a cocode element `c1` and a permutation `p` in the Mathieu group `Mat24` to a Parker loop automorphism. Here `c1` must be given in `cocode` representation. Permutation `p` must be given as a mapping `i -> p1[i], 0 <= i < 24`.

Up to sign, the image of an element of the Parker loop is the corresponding Golay code vector permuted by the permutation `p`. The sign of the image of the `i`-th positive basis vector of the Parker loop is given by bit `i` of `c1`. This determines the automorphism uniqely. We will write `AutPL(c1, p)` for that automorphism.

The function returns the automorphism `AutPL(c1, p)` as an array `m_out` of 12 integers of type `uint32_t`. The lowest 13 bits of `m_out[i]` contain the image of the `i`-th positive basis vector. Here each image is encoded as a Parker loop element as in function `mat24_mul_ploop`.

We also compute a quadratic form in the higher bits of the entries

of `m_out`, as described in function *mat24_autpl_set_qform()*. This facilitates computations in the automorphism group of the Parker loop.

Let `Id` be the neutral element in `Mat24`. Then we have

```
AutPL(c1, p)  = AutPL(c1, Id) * AutPL(0, p) .
```

void **`mat24_cocode_to_autpl`**(uint32_t c1, uint32_t *m_out)

Compute a diagonal Parker loop automorphism.

The function converts a cocode element `c1` to a Parker loop automorphism. Here `c1` must be given in `cocode` representation. Such an automorphism is called a diagnonal asutomorphism; it changes the signs of the Parker loop elements only.

The resulting automorphism is stored in `m_out` in the same way as in function *mat24_cocode_to_autpl()*.

If `p0` is an array representing the neutral element of the group `Mat24` then `mat24_cocode_to_autpl(cl, m_out)` is equivalent to `mat24_perm_to_autpl(c1, p0, m_out)`.

void **`mat24_autpl_to_perm`**(uint32_t *m1, uint8_t *p_out)

Extract permutation from Parker loop automorphism.

Ignoring the signs of the Parker loop, an automorphism `m1` of the Parker loop is an automorphism of the Golay code and can be represented as a permutation in the Mathieu group `Mat24`. Here `m1` must be encoded as described in function *mat24_perm_to_autpl()*.

The function computes the permutation in the group `Mat24` corresponding the automorphism `m1` and returns it in `p1` as a mapping `i -> p1[i]`.

uint32_t **`mat24_autpl_to_cocode`**(uint32_t *m1)

Extract cocode element from Parker loop automorphism.

Given an automorphism `m1` of the Parker loop, as constructed by function *mat24_perm_to_autpl()*, the function returns a cocode element `c` in `cocode` representation. Element `c` has the following property:

Let `p` be the permutation in `Mat24` obtained from `m1` by calling mat24_autpl_to_perm(m1, p). Then calling `mat24_perm_to_autpl(c, p, m2)`, where `m2` is a suitable array, constructs a copy `m2` of `m1` from `c` and `p`.

uint32_t **`mat24_op_ploop_autpl`**(uint32_t v1, uint32_t *m1)

Apply a Parker loop automorphism to a Parker Loop element.

Apply Parker loop automorphism `m1` to Parker Loop element `v1` and return the result as a Parker Loop element.

Here `m1` is a Parker loop autmorphism as constructed by function *mat24_perm_to_autpl()*. `v1` and the return value is an element of the Parker loop, encoded as in function *mat24_mul_ploop()*.

void **`mat24_mul_autpl`**(uint32_t *m1, uint32_t *m2, uint32_t *m_out)

Compute the product of two Parker Loop automorphisms.

Given two Parker Loop automorphism `m1, m2` the function computes `m1 * m2` and stores the result in `m_out`. All Parker loop automorphisms are encoded as in function *mat24_perm_to_autpl()*.

For an element a of the Parker loop we have `m_out(a) = m2(m1(a))`.

void **`mat24_inv_autpl`**(uint32_t *m1, uint32_t *m_out)

Compute the inverse of a Parker Loop automorphisms.

Given a Parker Loop automorphism `m1` the function computes the inverse of `m1` and stores the result in `m_out`. All Parker loop automorphisms are encoded as in function *mat24_perm_to_autpl()*.

void **mat24_perm_to_iautpl**(uint32_t c1, uint8_t *p1, uint8_t *p_out, uint32_t *m_out)

Compute inverse Parker Loop automorphism from permutation.

This is equivalent to

```
mat24_inv_perm(p1, p_out);
mat24_perm_to_autpl(c1, p1, m_temp);
mat24_inv_autpl(m_temp, m_out);
```

The function saves some intermedate steps so that it is faster.

void **mat24_perm_to_net**(uint8_t *p1, uint32_t *a_out)

Compute modified Benes network for permutation of 24 entries.

The Benes network is computed for the permutation `p:  i -> p1[i]`. The network consists of 9 layers. The returned array `a_out` of length 9 describes that network. In layer `i`, entry `j` is to be exchanged with entry `j + d[i]`, if bit `j` of the value `a_out[i]` is set. Here `d[i] = 1,2,4,8,16,8,4,2,1` for `i = 0,...,8`. For all such exchange steps we have `j & d[i] == 0`. We also assert that no entry with index >=24 will be touched.

void **mat24_op_all_autpl**(uint32_t *m1, uint16_t *a_out)

Auxiliary function for computing in the monster.

The function is used for applying the automorphism `m1` of the Parker loop to a vector of the `196884`-dimensional representation of the monster. `m1` is encoded as in function *mat24_perm_to_autpl()*.

It computes a table `a_out[i], i = 0,...,0x7ff`, such that `(a_out[i] & 0x7ff)` is the image `m1(i)` of the Parker loop element `i` modulo the center of the Parker loop. Signs are stored in bits `12,...,14` of `a_out[i]` as follows:

```
Bit 12: (sign of m1(i)) ^ (odd &  P(i))

Bit 13: (sign of m1(i))

Bit 14: (sign of m1(i)) ^ (bit 11 of m1(i))
```

Here `odd` is the parity of the automorphism, and `P()` is the power map of the Parker loop.

void **mat24_op_all_cocode**(uint32_t c1, uint8_t *a_out)

Auxiliary function for computing in the monster.

This is a simplified version of function *mat24_op_all_autpl()*, which is used for applying the diagonal automorphism `c1` of the Parker loop (encoded in `cocode` representation) to a vector of a representation of the monster.

The function computes a table `a_out[i], i= 0,...,0x7ff`, containing the signs related to this operation as follows:

```
 Bit 0:  (sign of c1(i)) ^ (odd &  P(i))

 Bit 1:  (sign of c1(i))

 Bit 2:  same as bit 1
```

Here `odd` and `P()` are as in function *mat24_op_all_autpl()*.

C functions for generating random elements of M_{24}

File `mat24_random.c` contains the C implementations of the functions for generation random elements of some subgroups of the Mathieu group M_{24}.

Equivalent python function are coded in module `mmgroup.tests.test_mat24.test_mat24_rand`.

A subgroup of M_{24} is decribed by an integer of type `uint_32_t` encoding a bit mask. Each bit in that mask encodes a certain subgroup of M_{24}. By combining several bits with bitwise pr we may evcode the intersection of the subgroups corresponding to the bits being set.

The mapping of the bits to the subgroups is given in the description of the `enum` type `mat24_rand_flags` in file `mat24_functions.h`.

Functions

uint32_t **`mat24_complete_rand_mode`**(uint32_t u_mode)

Complete an intersection of subgroups of M_{24}.

Here the integer `u_mode` is a combination of flags of type `enum mat24_rand_flags` describing an intersection H of subgroups of M_{24}. Then the group H may be contained in more subgroups of M_{24} encoded as bits of an integer of type `enum mat24_rand_flags`. This function sets all bits in `u_mode` corresponding to groups containing H. Furthermore, the function clears all unused bits in parameter `u_mode`.

The function returns the modified parameter `u_mode`.

int32_t **`mat24_perm_in_local`**(uint8_t *p1)

Compute some subgroups containing an element of M_{24}.

Let p_1 be a permutation in M_{24} given as an array of 24 integers. The function computes a set of subgroups of M_{24} containing p_1. These computations are done for all subgroups corresponding to the flags defined in `enum mat24_rand_flags`. The function returns an integer `mode` that is the combination of flags of type `enum mat24_rand_flags` describing the subgroups of M_{24} containing H .

The function returns -1 if p_1 is not in M_{24}.

int32_t **`mat24_perm_rand_local`**(uint32_t u_mode, uint32_t u_rand, uint8_t *p_out)

Generate a random element of a subgroup of M_{24}.

The function generates an element of a subgroup H of the Mathieu group M_{24}. Here the parameter `u_mode` is a combination of flags of type `enum mat24_rand_flags` describing the group H as an intersection of subgroups of M_{24}. The generated permutation is stored in the array `p_out` of length 24.

Parameter `u_rand` is an integer describing the element of subgroup H to be generated. Here `u_rand` is reduced modulo the order of H. In order to generate a uniform random element of H, the user should generate a uniform random number `0 <= u_rand < MAT24_ORDER`, where `MAT24_ORDER` is the order of the group M_{24}.

The function returns 0 in case of success and -1 in case of failure.

int32_t **`mat24_m24num_rand_local`**(uint32_t u_mode, uint32_t u_rand)

Generate number of random element of a subgroup of M_{24}.

The function generates an element of a subgroup H of the Mathieu group M_{24}. Here the parameters `u_mode` and and `u_rand` are as in function `mat24_perm_rand_local`.

The function returns the number of the generated element of M_{24} in case of success and -1 in case of failure.

See function `mat24_m24num_to_perm` for the numbering of the elements of M_{24}.

int32_t **mat24_m24num_rand_adjust_xy**(uint32_t u_mode, uint32_t v)

Make an element of the Parker loop compatible with a subgroup.

Here parameter `v` is an element d of the Parker loop encoded as in file `mat24.c`. Parameter `u_mode` describes a subgroup of the Mathieu group M_{24} as in function `mat24_perm_rand_local`.

Eventually, we want to construct random elements in a larger group then M_{24}. For some values of `u_mode` we want to use additional generators corresponding to a subloop of the Parker loop. Here the details are dictated by the 2-local structure of the Monster.

If `u_mode` is set so that one or more Golay cocode vectors of weight 2 are fixed pointwise then we require the scalar product of d with all these fixed weight-2 vectors to be zero. The function modifies the Parker loop element d appropriately and returns the modified element.

3.2.13 Generating C code for the `mmgroup.mat24` extension

In this section we give a brief overview over the modules in `mmgroup.dev.mat24` used for generating C code for the `mmgroup.mat24` extension.

Module `mat24_ref`

Module `mat24_ref` contains the class `Mat24.`

Class `Mat24` in module `mmgroup.dev.mat24.mat24_ref` is the table-providing class used by the code generator to generate the C file `mat24_functions.c`. That C file contains the functionality of the python extension `mmgroup.mat24`. Class `Mat24` may also be used as a table-providing class for generating other C files.

Class `Mat24` also exports the same functions as the `mmgroup.mat24` extension as class methods. So it can be used for testing that extension and also as a pure-python substitute for that extension.

Class `Mat24` is based on class `mat24tables.Mat24Tables`. It also uses classes, functions and tables from the following modules in `mmgroup.dev.mat24`:

- `make_addition_table`, `make_mul_transp`, `mat24aux`, `mat24heptad`, `mat24theta`

Module `mat24tables`

Module `mat24tables` contains the class `Mat24Tables`

Class `Mat24Tables` is a base for class `Mat24` in module `mmgroup.dev.mat24.mat24_ref`. It contains the basis of the Golay code and of (a set of representatives of) the Golay cocode. It also contains tables for fast conversion of a vector in `V = GF(2)**24` from the standard to the internal representation and vice versa. That class also contains a table of the syndromes of all `2048` odd elements of the cocode. The `759` octads, i.e. Golay code word of length `8` are numbered from `0` to `758`. Class `Mat24Tables` provides tables for computing the number of a octad from a Golay code word representing an octad and vice versa.

Class `Mat24Tables` also contains python versions of some C functions in file `mat24_functions.c` using these tables.

Module mat24aux

The module contains classes Lsbit24Function and MatrixToPerm.

Class Lsbit24Function contains tables and directives for computing the least significant bit of a 24-bit integer using a DeBruijn sequence. This may be overkill for such a simple function, but it may also be considered as a didactic example of a table-providing class.

Class MatrixToPerm contains a directive that generates highly optimized code for converting a bit matrix acting on a Golay code word to a permutation in the Mathieu group Mat_24. Here we assume the bit matrix actually encodes an element of the Mathieu group; otherwise garbage is returned. The Golay code word must be given as an integer in gcode representation.

Module mat24heptad

Support for completing a permutation in the Mathieu group

A *heptad* is s subset of seven elements of the set on which that Mathieu group acts. Under certain circumstances a mapping from one heptad to another heptad can be completed to a unique element of mat_24. The C function mat24_perm_from_heptads in file mat24_functions.c performs that task. It calls a simpler function mat24_perm_complete_heptad in that file which completes a mapping from a fixed heptad to variable heptad.

This file contains python implementations of functions mat24_perm_complete_heptad and mat24_perm_from_heptads.

The C function mat24_perm_to_m24num maps the elements of the Mathieu group Mat_24 to the set if integers 0, ..., order(Mat_24). Function mat24_perm_complete_heptad is also used as a subroutine of function mat24_m24num_to_perm which compute the inverse of that mapping. This module also contains python implementations of these functions.

Class HeptadCompleter is a table-providing class that is used for code generation. It also exports the functionality of this module.

Function hint_for_complete_heptads prints an explanation of the implementation of function mat24_perm_complete_heptad, including some precalulated data required for that implementation.

Function test_complete_octad test the correcteness of function mat24_perm_complete_heptad. Here it suffices to check that the identity permutation is generated correctly from the required input values, and that the intersections of (possibly complemented) octads and syndromes are indeed singletons if the ought to be singletons.

Module make_addition_table

The module contains class BitMatrixMulFix.

Class BitMatrixMulFix contains a directive that generates code to multiply a fixed bit matrix with a variable bit matrix.

The C function mat24_perm_to_matrix uses this kind of matrix multiplication to convert a permutation in the Mathieu group to a bit matrix operating on a Golay code word.

Module `make_mul_transp`

The module contains class `BitMatrixMulTransp`.

Class `BitMatrixMulTransp` contains a directive that generates code to multiply a bit vector with the transposed matrix of a fixed bit matrix. Depending on the size of the matrix and the bit length of the underlying integer type, several bit vectors can be multiplied with the same matrix simultaneously.

The C function `mat24_autpl_set_qform` uses that matrix multiplication, see section *Implementing Automorphisms of the Parker loop* in the *Guide for developers* for background.

3.3 Description of the `mmgroup.generators` extension

Module `generators` contains the definition of the generators of the monster, so that they may be used in C files. It also contains support for the subgroups N_0 of structure $2^{2+11+2\cdot 11}.(\mathrm{Sym}_3 \times M_{24})$ and G_{x0} of structure $2^{1+24}.\mathrm{Co}_1$ of the monster, as described in [Con85] and [Sey20]. Here $M24$ is the Mathieu group acting on 24 elements, and Co_1 is the automorphism group of the 24-dimensional Leech lattice modulo 2.

Here we fully support the computation in the subgroup N_0 based on the generators defined in this module, so that a word in the generators of N_0 can easily be reduced to a standard form.

We also support the operation of the group G_{x0} on the Leech lattice mod 2 and mod 3 (in some cases up to sign only). For a full support of the subgroup G_{x0} we also have to compute in a Clifford group, which is implemented in module `clifford12`.

Our set of generators of the monster group is defined in section *The Monster group*. The C implementation of this set of generators is defined in section *Header file mmgroup_generators.h*.

The intersection $N_0 \cap G_{x0}$ is a group N_{x0} of structure $2^{1+24}.2^{11}.M_{24}$. The group M_{24} (and, to some extent, also the group N_{x0}) is supported by the C functions in file `mat24_functions.c`.

3.3.1 The Leech lattice and the extraspecial group Q_{x0}

Let Q_{x0} the normal subgroup of G_{x0} of structure 2^{1+24}. Then Q_{x0} is an extraspecial 2 group and also a normal subgroup of G_{x0}. Let Λ be the Leech lattice. The quotient of Q_{x0} by its center $\{\pm 1\}$ is isomorphic to $\Lambda/2\Lambda$, which is the Leech lattice modulo 2.

For $e_i \in Q_{x0}$ let $\tilde{e}_i$ be the vector in the Leech lattice (mod 2) corresponding to $\pm e_i$. Then $e_1^2 = (-1)^s$ for $s = \langle \tilde{e}_1, \tilde{e}_1 \rangle /2$, where $\langle ., . \rangle$ is the scalar product in the Leech lattice. For the commutator $[e_1, e_2]$ we have $[e_1, e_2] = (-1)^t$, $t = \langle \tilde{e}_1, \tilde{e}_2 \rangle$.

3.3.2 Leech lattice encoding of the elements of Q_{x0}

An element of of Q_{x0} can be written uniquely as a product $x_d \cdot x_\delta$, $d \in \mathcal{P}, \delta \in \mathcal{C}^*$, see [Sey20], section 5. Here $\mathcal{P}$ is the Parker loop and $\mathcal{C}^*$ is the Golay cocode. We encode the element $x_d \cdot x_\delta$ of Q_{x0} as an integer x as follows:

$$x = 2^{12} \cdot d \oplus (\delta \oplus \theta(d)) \, .$$

Here elements of the Parker loop and elements of the cocode are encoded as integers as in section *The Parker loop* and *The Golay code and its cocode*. θ is the cocycle given in section *The basis of the Golay code and of its cocode*, and '$\oplus$' means bitwise addition modulo 2. Note that a Parker loop element is 13 bits long (with the most significant bit denoting the sign) and that a cocode element is 12 bits long.

From this representation of Q_{x0} we obtain a representation of a vector in the Leech lattice modulo 2 by dropping sign bit, i.e. the most significant bit at position 24. A vector addition in the Leech lattice modulo 2 can be done by applying the XOR operator `^` to the integers representing the vectors, ignoring the sign bit.

3.3.3 Special elements of the group Q_{x0}

We write Ω for the positive element of the Parker loop such that $\tilde{\Omega}$ is the Golay code word $(1, \ldots, 1)$ as in [Con85] and [Sey20]. In this specifiction we also write Ω for the element x_Ω of Q_{x0} and for the element $\tilde{x}_\Omega$ of the Leech lattice modulo 2 if the domain of Ω is clear from the context. Then Ω has Leech lattice encoding `0x800000` in our chosen basis of the Golay code; and the element Ω of $\Lambda/2\Lambda$ corresponds to the standard coordinate frame of the real Leech lattice.

For fast computations in the monster group it is vital to compute in the centralizer of a certain short element x_β of Q_{x0}, where β is an even coloured element of the Golay cocode, as described in [Sey20]. Here we choose the cocode element β corresponding to the element $(0, 0, 1, 1, 0, \ldots, 0)$ of GF_2^{24}. Then the centralizer of x_β is isomorphic to the a double cover baby monster group and contains the generators τ and ξ of the monster. We also write β for the element x_β of Q_{x0} and for the element $\tilde{x}_\beta$ of $\Lambda/2\Lambda$ in the same way as for the element Ω. Then β has Leech lattice encoding `0x200` in our chosen basis.

Module `gen_leech_reduce.c` contains functions for rotating arbitrary type-4 vectors in $\Lambda/2\Lambda$ to Ω and for rotating arbitrary type-2 vectors in $\Lambda/2\Lambda$ to β.

3.3.4 Computations in the Leech lattice modulo 3

For the construction of the subgroup G_{x0} of the monster we also require the automorphism group Co_0 of the **real** Leech lattice, as decribed in [Con85]. That group has a faithful representation as an automophism group of $\Lambda/3\Lambda$, but not of $\Lambda/2\Lambda$. Module `gen_leech3.c` provides functions for computing in the Leech Lattice modulo 3.

Note that in [Sey20] the operation of the generators $x_d, y_d, x_\delta, x_\pi, \xi$ of G_{x0} is also defined on the real Leech lattice.

3.3.5 Header file mmgroup_generators.h

The header file `mmgroup_generators.h` contains definitions for the C files in the `generator` extension. This extension comprises files `mm_group_n.c`, `gen_xi_functions.c`, and `gen_leech.c`.

In this header we also define an `enum MMGROUP_ATOM_TAG_` that specifies the format of an atom that acts as a generator of the monster group.

Defines

`MMGROUP_ATOM_TAG_ALL`

Tag field of a monster group atom

`MMGROUP_ATOM_DATA`

Data field of a monster group atom

`gen_leech2_def_mul`(x1, x2, result)

Macro version of function `gen_leech2_mul`.

Macro *`gen_leech2_def_mul(x1, x2, result)`* is equivalent to the statement `result = gen_leech2_mul(x1, x2)`. The macro generates a sequence of statements!

Caution:

Here `result` must be an integer lvalue that is different from both integers, `x1` and `x2`!

Enums

enum **`MMGROUP_ATOM_TAG_`**

In this header file we also define the tags for the atoms generating the Monster group. An element of the monster group is represented as an array of integers of type `uint32_t`, where each integer represents an atom, i.e. an atomic element of the monster. An atom represents a triple `(sign, tag, value)`, and is encoded in the following bit fields of an unsigned 32-bit integer:

```
Bit 31  | Bit 30,...,28  | Bit  27,...,0
--------|----------------|----------------
 Sign   | Tag            | Value
```

Standard tags and values are defined as in the constructor of the Python class `mmgroup.mm`, see section **The monster group** in the **API reference**. If the `sign` bit is set, this means that the atom bit given by the pair `(tag, value)` has to be inverted. In ibid., a tag is given by a small letter. These small letters are converted to 3-bit numbers as follows:

```
Tag  | Tag number | Range of possible values i
-----|------------|---------------------------
 'd' |     1      |  0 <= i < 0x1000
 'p' |     2      |  0 <= i < 244823040
 'x' |     3      |  0 <= i < 0x2000
 'y' |     4      |  0 <= i < 0x2000
 't' |     5      |  0 <= i < 3
 'l' |     6      |  0 <= i < 3
```

A tag with tag number 0 is interpreted as the neutral element. A tag with tag number 7 is illegal (and reserved for future use).

Tags with other letters occuring in the constructor of class `MM` are converted to a word of atoms with tags taken from the table above.

For tags ‘t’ and ‘l’ the values 0 <= i <= 3 are legal on input.

Values:

enumerator **`MMGROUP_ATOM_TAG_1`**

Tag indicating the neutral element of the group

enumerator **`MMGROUP_ATOM_TAG_I1`**

Tag indicating the neutral element of the group

enumerator **`MMGROUP_ATOM_TAG_D`**

Tag corresponding to ‘d’

enumerator **`MMGROUP_ATOM_TAG_ID`**

Tag corresponding to inverse of tag ‘d’

enumerator **MMGROUP_ATOM_TAG_P**

Tag corresponding to 'p'

enumerator **MMGROUP_ATOM_TAG_IP**

Tag corresponding to inverse of tag 'p'

enumerator **MMGROUP_ATOM_TAG_X**

Tag corresponding to 'x'

enumerator **MMGROUP_ATOM_TAG_IX**

Tag corresponding to inverse of tag 'x'

enumerator **MMGROUP_ATOM_TAG_Y**

Tag corresponding to 'y'

enumerator **MMGROUP_ATOM_TAG_IY**

Tag corresponding to inverse of tag 'y'

enumerator **MMGROUP_ATOM_TAG_T**

Tag corresponding to 't'

enumerator **MMGROUP_ATOM_TAG_IT**

Tag corresponding to inverse of tag 't'

enumerator **MMGROUP_ATOM_TAG_L**

Tag corresponding to 'l'

enumerator **MMGROUP_ATOM_TAG_IL**

Tag corresponding to inverse of tag 'l'

3.3.6 C functions implementing the group N_0

We describe an implementation of the subgroup N_0 of the monster group.

The subgroup N_0 of the monster group of structure $2^{2+11+2\cdot 11}.(\mathrm{Sym}_3 \times \mathrm{M}_{24})$ has been described in [Con85]. Theorem 5.1 in [Sey20] reduces the group operation in N_0 to easy calculations in the Parker loop $\mathcal{P}$, the Colay cocode $\mathcal{C}^*$, and the group $\mathrm{Aut}_{\mathrm{St}}\mathcal{P}$ of standard automorphisms of $\mathcal{P}$. The loops $\mathcal{P}$, $\mathcal{C}^*$, and $\mathrm{Aut}_{\mathrm{St}}\mathcal{P}$ are described in section *Basic structures*. Module `mat24_functions.c` provides the required functions for computing in these loops.

Using the notation in section *The Monster group* we may describe an element of N_0 as a product:

$$\tau^t y_f x_e x_\delta x_\pi \ ; \quad 0 \le t < 3; \ e, f \in \mathcal{P}; \ \delta \in \mathcal{C}^*; \ \pi \in \mathrm{Aut}_{\mathrm{St}}\mathcal{P} \ .$$

This representation is unique if we require f to be in a transversal of $\mathcal{P}/Z(\mathcal{P})$ and π to be a standard representative in $\mathrm{Aut}_{\mathrm{St}}\mathcal{P}$ as described in section *Automorphisms of the Parker loop*.

We store an element of N_0 an a quintuple t, f, e, δ, π of five integers of type `uint32_t`. For f, e we use the numbering in class `PLoop` in module `mmgroup`; for δ we use the numbering in class `Cocode` in module `mmgroup`. Here π refers to a standard representative in $\mathrm{Aut}_{\mathrm{St}}\mathcal{P}$. These standard representatives correspond to the elements of the Mathieu group M_{24}; and we use the numbering of the elements of M_{24} described in class `AutPL|` in module `mmgroup`.

Most functions in module `mm_group_n.c` take a pointer to a 5-tuple (t, f, e, δ, π) representing an element g of the N_0 as their first argument. Then the tuple representing g is modified to a tuple representing an element $g_2 = g \cdot g_1$, with the element g_1 of N given by one or more subsequent arguments of the function.

The functions in module `mm_group_n.c` may cause some overhead due to the fact that element of the Mathieu group M_{24} is represented as an integer. But compared to an operation of the monster group on its 196884-dimensional that overhead is negligible.

C interface for file mm_group_n.c

The functions in file `mm_group_n.c` implement the subgroup N_0 of structure $2^{2+11+2\cdot 11}.(\mathrm{Sym}_3 \times \mathrm{Mat}_{24})$ on the monster group. Elements of N_0 are represented as arrays of five integers of type `uint32_t` as described in the document *The C interface of the mmgroup project*.

Functions

void **`mm_group_n_mul_delta_pi`**(uint32_t *g, uint32_t delta, uint32_t pi)

Multiply $g \in N_0$ with $x_\delta x_\pi$.

Put $g = g x_\delta x_\pi$. Here the integer δ represents an element of the Golay cocode and π is the number of an element of M_{24} corresponding to a standard representative in the automorphism group $\mathrm{AUT}(\mathrm{PL})$ of the Parker loop.

g is given as an array of five 32-bit integers.

void **`mm_group_n_mul_inv_delta_pi`**(uint32_t *g, uint32_t delta, uint32_t pi)

Multiply $g \in N_0$ with $(x_\delta x_\pi)^{-1}$.

Put $g = g \cdot (x_\delta x_\pi)^{-1}$. Here g, x_δ, and x_π are as in function `mm_group_n_mul_delta_pi`.

void **`mm_group_n_mul_x`**(uint32_t *g, uint32_t e)

Multiply $g \in N_0$ with x_e.

Put $g = g \cdot x_e$. Here the integer e represents an element of the Parker loop.

g is given as an array of five 32-bit integers.

void **`mm_group_n_mul_y`**(uint32_t *g, uint32_t f)

Multiply $g \in N_0$ with y_f.

Put $g = g \cdot y_f$. Here the integer f represents an element of the Parker loop.

g is given as an array of five 32-bit integers.

void **`mm_group_n_mul_t`**(uint32_t *g, uint32_t t)

Multiply $g \in N_0$ with the triality element.

Put $g = g \cdot \tau^t$, where τ is the triality element. g is given as an array of five 32-bit integers.

void **`mm_group_n_clear`**(uint32_t *g)

Set $g \in N_0$ to the value of the neutral element.

Put $g = 1$. Element g of N_0 is given as an array of five 32-bit integers.

void **`mm_group_n_copy_element`**(uint32_t *g_1, uint32_t *g_2)

Copy the element g_1 of N_0 to g_2.

Elements g_1, g_2 of N_0 are given as arrays of five 32-bit integers.

void **mm_group_n_mul_element**(uint32_t *g_1, uint32_t *g_2, uint32_t *g_3)

Multiply elemnts of the group N_0.

Put $g_3 = g_1 \cdot g_2$. Elements g_1, g_2, g_3 of N_0 are given as an array of five 32-bit integers.

These arrays may overlap.

void **mm_group_n_mul_inv_element**(uint32_t *g_1, uint32_t *g_2, uint32_t *g_3)

Multiply $g_1 \in N_0$ with $g_2^{-1} \in N_0$.

Put $g_3 = g_1 \cdot g_2^{-1}$. Elements g_1, g_2, g_3 of N_0 are given as arrays of five 32-bit integers.

These arrays may overlap.

void **mm_group_n_inv_element**(uint32_t *g_1, uint32_t *g_2)

Invert an element g_1 of N_0.

Put $g_2 = g_1^{-1}$. Elements g_1, g_2 of N_0 are given as arrays of five 32-bit integers.

These arrays may overlap.

void **mm_group_n_conjugate_element**(uint32_t *g_1, uint32_t *g_2, uint32_t *g_3)

Conjugate $g_1 \in N_0$ with $g_2 \in N_0$.

Put $g_3 = g_2^{-1} \cdot g_1 \cdot g_2$. Elements g_1, g_2, g_3 of N_0 are given as arrays of five 32-bit integers.

These arrays may overlap.

uint32_t **mm_group_n_mul_word_scan**(uint32_t *g, uint32_t *w, uint32_t n)

Multiply $g \in N_0$ with an element of the monster.

Let `w` be a word of generators of the monster group of length `n`. Let `k` be the greatest number such that all prefixes of `w` of length at most `k` are in the group N_0. Let a be the element of N_0 corresponding to the prefix of `w` of length `k`.

Let $g \in N_0$ is given as an array `g` of five 32-bit integers.

Then the function replaces the value g in the array `g` by $g \cdot a$. It returns the number `k` of atoms of the word `w` processed.

uint32_t **mm_group_n_mul_atom**(uint32_t *g, uint32_t atom)

Multiply $g \in N_0$ with an atom.

Put $g = g \cdot a$. Here a is the generator of the group N_0 given by parameter `atom` as described in the header file `mmgroup_generators.h`.

g is given as an array of five 32-bit integers.

The function returns 0 in case of success and the (possibly simplified) atom in case of failure.

uint32_t **mm_group_n_scan_word**(uint32_t *w, uint32_t n)

Scan word of generators of the monster for membership in N_0.

Let `w` be a word of generators of the monster group of length `n`. The function returns the greatest number `k` such that all prefixes of `w` of length at most `k` are in the group N_0.

uint32_t **mm_group_n_conj_word_scan**(uint32_t *g, uint32_t *w, uint32_t n)

Conjugate $g \in N_0$ with an element of the monster.

Let `w` be a word of generators of the monster group of length `n`. Let `k` be the greatest number such that all prefixes of `w` of length at most `k` are in the group N_0. Let a be the element of N_0 corresponding to the prefix of `w` of length `k`.

Let $g \in N_0$ is given as an array `g` of five 32-bit integers.

Then the function replaces the value g in the array `g` by $a^{-1} \cdot g \cdot a$. It returns the number `k` of atoms of the word `w` processed.

uint32_t **mm_group_n_reduce_element**(uint32_t *g)

Reduce $g \in N_0$ to a standard form.

The representation of g is reduced to a standard form.

Technically, we reduce the product $y_f x_e$ to $y_{f'} x_{e'}$, such that $0 \leq f' <$ 0x800 holds. If x_e is in the center of the Parker loop, but x_f is not in that center then we put $x_{e'} = 0$.

g is given as an array of five 32-bit integers.

uint32_t **mm_group_n_reduce_element_y**(uint32_t *g)

Reduce $g \in N_0$ to a standard form.

The representation of g is reduced to a standard form.

Here we reduce the product $y_f x_e$ to $y_{f'} x_{e'}$, such that $0 \leq f' <$ 0x800 **always** holds.

g is given as an array of five 32-bit integers.

uint32_t **mm_group_n_to_word**(uint32_t *g, uint32_t *w)

Convert $g \in N_0$ to a word of generators.

The representation of g is converted to a word of generators of the monster group. The entries of that word are stored in the buffer referred by parameter `w`. The entries of that word are encoded as described in file `mmgroup_generators.h`. Word `w` may have up to five entries. The function returns the length of the word `w`.

g is given as an array of five 32-bit integers.

The element g is reduced with function `mm_group_n_reduce_element`.

It is legal to put `w` = `g`.

uint32_t **mm_group_n_right_coset_N_x0**(uint32_t *g)

Map $g \in N_0$ to an element of N_{x0}.

The function changes the element g of N_0 to an element g' of N_{x0} and returns an exponent $0 \leq e < 3$ such that $g = g' \cdot \tau^e$.

uint32_t **mm_group_n_to_word_std**(uint32_t *g, uint32_t *w)

Convert $g \in N_0$ to a standard word of generators.

The representation of g is converted to a word of generators of the monster group in the standard order. The entries of that word are stored in the buffer referred by parameter `w`.

The standard order of the generators of the monster group in the reduced representation of an element of N_0 differs from the order of the generators returned by function `mm_group_n_to_word`. Apart from this difference the action of this function is the same as in that function.

It is legal to put `w` = `g`.

int32_t **mm_group_n_conj_to_Q_x0**(uint32_t *g)

Transform element of N_0 to an element of Q_{x0}.

Let $g \in N_x 0$ be stored in the array `g` of five 32-bit integers.

The function tries to calculate a number $0 \leq e < 3$ and an element q of the subgroup Q_{x0} of N_0 with $g = \tau^{-e} q \tau^e$. Here τ is the triality element in N_0.

In case of succes we return the element q in bits 24,. . . ,0 of the return value in **Leech lattice encoding** and we return the number e in bits 26,. . . ,25 of the return value. In case of failure we return -1.

uint32_t **mm_group_split_word_n**(uint32_t *word, uint32_t length, uint32_t *g)

Split an element of N_0 from a word of generators.

Given a array `word` of generators of the monster group of a given `length`, that word is split into a possibly shorter word `word1` and an element `g` of the group N_0 such that `word` = `word1` * `g`. Then `word1` is a prefix of `word`. The function returns the length of the prefix `word1` of `word`. It does not change `word`. Here we just scan the `word` from the right, checking for atoms ordered in a way compatible to our representation of N_0.

Words of generators of the monster are implemented as described in file `mmgroup_generators.h`. Output g is given as an array of five 32-bit integers.

uint32_t **mm_group_mul_words**(uint32_t *w1, uint32_t l1, uint32_t *w2, uint32_t l2, int32_t e)

Multiply a word of generators of the monster.

Given a word `w1` of length `l1` and a word `w2` of length `l2` of generators of the monster group, we compute the product `w3 = w1 * w2**e`. Here `**` means exponentiation; negative exponents are supported. The word `w1` is replaced by the word `w3`. The funcion returns the length of the word `w3`.

A word representing `w2**e` is appended to word `w1` and then the result is simplified using the relations inside the group N_0. The result `w3` is reduced (with respect to these relations) if input `w1` is reduced.

Caution!

The buffer for the word `w1` referred by pointer `w1` must be able to store at least `l1 + 2 * abs(e) * l2` entries of type `uint32_t`. Here the user must provide sufficient space!

Words of generators of the monster are implemented as described in file `mmgroup_generators.h`.

void **mm_group_invert_word**(uint32_t *w, uint32_t l)

Invert a word of generators of the monster.

Given a word `w` of length `l` the function changes the word in the buffer `w` to its inverse in place.

Words of generators of the monster are implemented as described in file `mmgroup_generators.h`.

uint32_t **mm_group_check_word_n**(uint32_t *w1, uint32_t l1, uint32_t *g_out)

Check if a words of generators of the monster is in N_0.

We check if the word `w1` of length `l1` of generators of the monster group is in the subroup N_0. If this is the case then we store the word `w1` as an element of N_0 in the array `g_out` of five 32-bit integers as desribed above. The function returns the following status information:

0: `w1` is the neutral element the monster group

1: `w1` is in N_0, but not the neutral element

2: `w1` is not in N_0

3: Nothing is known about `w1`

We check the relations in the generators N_0 only. The output in `g_out` is valid only if the function returns 0 or 1.

Words of generators of the monster are implemented as described in file `mmgroup_generators.h`.

uint32_t **mm_group_words_equ**(uint32_t *w1, uint32_t l1, uint32_t *w2, uint32_t l2, uint32_t *work)

Check if two word of generators of the monster are equal.

We check if the word `w1` of length `l1` of generators of the monster group is equal to the word `w2` of length `l2` of generators. The function returns the following status information:

0: `w1` == `w2`

1: `w1` != `w2

Greater than 1: equality of `w1` and `w2` is not known

If the function cannot check the equality of `w1` and `w2` then it computes a word `w3` of length `l3` of generators of the monster group such that `w1 == w2` if and only if `w3` is the neutral element. Then the function stores the word `w3` in the buffer `work` and it returns `l3 + 2`.

Buffer `work` must have size at least `max(2 * l1, l1 + 2 * l2)`.

Words of generators of the monster are implemented as described in file `mmgroup_generators.h`.

3.3.7 C functions for the operation of G_{x0} on the Leech lattice

The functions in file `gen_leech.c` implement the operation of the subgroup G_{x0} (of structure $2^{1+24}.\mathrm{Co}_1$) of the monster group on its extraspecial subgroup Q_{x0} (of structure 2^{1+24}) by conjugation.

Here an element of the group Q_{x0} is given as a 25-bit integer in Leech lattice encoding. From that encoding we obtain an encoding of $\Lambda/2\Lambda$ (where Λ is the Leech lattice) by dropping the most significant bit. This corresponds to the isomorphism $Q_{x0}/\{\pm 1\} \cong \Lambda/2\Lambda$.

An element of the group G_{x0} is encoded as an array of 32-bit integers, where each integer corresponds to a generator of the group, as described in the documentation of the header file `mmgroup_generators.h`.

A vector in the Leech lattice modulo 2 has a type and also a subtype as described in **The mmgroup guide for developers**, section *Computations in the Leech lattice modulo 2*. Here the subtype is a two-digit decimal number, where the first digit is the type. Function `gen_leech2_subtype` returns the subtype as a BCD-coded integer. E.g. the subtype 46 is returned as the hexadecimal integer 0x46. So the type can be obtained from the subtype via a shift operation.

For computations in the group G_{x0} or Co_1 it is important to find an element of G_{x0} that maps an arbitrary type-4 vector in $\Lambda/2\Lambda$ to the unique type-4 vector Ω, see *Computations in the Leech lattice modulo 2*. Function `gen_leech2_reduce_type4` performs this task.

The construction of the group G_{x0} also requires some computations in the automorphism group Co_0 (of structure $2.\mathrm{Co}_1$) of Λ, see [Con85], section 9 or [Sey20], section 9. The group Co_0 acts faithfully on $\Lambda/3\Lambda$. Therefore file `gen_leech.c` also provides some functions for dealing with vectors in the Leech lattice modulo 3.

For any $w \in \Lambda$ the vector $v = \sqrt{8} \cdot w$ has integral coordinates. We represent a vector $w \in \Lambda/3\Lambda$ by the vector $v = \sqrt{8} \cdot w$, with the coordinates of v taken modulo 3. We represent each coordinate $v_i, 0 \leq i < 24$ as a two-bit integer; so all 48 bits of v fit into the a 64-bit integer. Let $v_{i,1}, v_{i,0}$ be the bits of v_i of valence 2^1 and 2^0, respectively. Then we encode v as an 48-bit integer x so that bit $24 \cdot j + i$ of x is equal to $v_{i,j}$. This encoding looks peculiar, but it greatly simplifies the interaction with the functions in file `mat24_functions.c`. We call this encoding the **Leech lattice mod 3** encoding. On input, both bits $v_{i,1}, v_{i,0}$ may be equal to 1; but on output, at most on of these bits is equal to 1.

In [Sey20], section 9.3, the generators $x_d, x_\delta, y_\delta, x_\pi, \xi$ of G_{x0} are also defined as generators of a group G_{x1} with $|G_{x1} : G_{x0}| = 2$. The group G_{x1} operates on Λ, but the group G_{x0} does not. So we take these generators as generators of G_{x1}. The kernel of the operation of G_{x1} on Λ is the group Q_{x0} generated by the elements $x_d, x_\delta, y_{-1}, d \in \mathcal{P}, \delta \in \mathcal{C}$. Here the element y_Ω acts as the central element -1.

C interface for file gen_leech.c

The functions in file `gen_leech.c` implement operations on the vectors of the Leech lattice modulo 2 and on the subgroup Q_{x0}. We use the terminology defined in the document *The C interface of the mmgroup project*, section *Description of the mmgroup.generators extension*.

Functions

uint32_t **`gen_leech2_mul`**(uint32_t x1, uint32_t x2)

Return product of two elements the group Q_{x0}.

Here all elements of the group Q_{x0} are encoded in **Leech lattice encoding**. The function returns the product of the elements `x1` and `x2` of Q_{x0}.

uint32_t **`gen_leech2_pow`**(uint32_t x1, uint32_t e)

Return power of element the group Q_{x0}.

Here all elements of the group Q_{x0} are encoded in **Leech lattice encoding**. The function returns the power `x1**e` of the element `x1` of Q_{x0}.

uint32_t **`gen_leech2_scalprod`**(uint32_t x1, uint32_t x2)

Return scalar product in the Leech lattice modulo 2.

Here all elements of Leech lattice modulo 2 are encoded in Leech lattice encoding**. The function returns the scalar product of the vectors `x1` and `x2` in the Leech lattice modulo 2, which may be 0 or 1.

uint32_t **`gen_leech2_op_word`**(uint32_t q0, uint32_t *g, uint32_t n)

Perform operation of G_{x0} on Q_{x0}.

The function returns the element $g^{-1}q_0g$ for $q_0 \in Q_{x0}$ and $g \in G_{x0}$. Here g is given as a word of genenators of length n in the array `g`. Each atom of the word g is encoded as defined in the header file `mmgroup_generators.h`. Parameter q_0 and the result are encoded in **Leech lattice encoding**.

The function succeeds also in case $g \notin G_{x0}$ if $h^{-1}q_0h \in G_{x0}$ for all prefixes h of g.

uint32_t **`gen_leech2_op_atom`**(uint32_t q0, uint32_t g)

Atomic operation of G_{x0} on Q_{x0}.

Equivalent to `gen_leech2_op_word(q0, &g, 1)`.

uint32_t **`gen_leech2_prefix_Gx0`**(uint32_t *g, uint32_t len_g)

Scan prefix in G_{x0} of a word in the monster group.

Let $g \in G_{x0}$ be stored in the array `g` of length `len_g` as a word of generators of the subgroup G_{x0} of the monster. The function returns the maximum length `len_g` such that every prefix of the word in `g` of length `<= len_g` is in the group G_{x0}.

uint32_t **`gen_leech2_op_word_many`**(uint32_t *q, uint32_t m, uint32_t *g, uint32_t n)

Perform operation of G_{x0} on elements of Q_{x0}.

Let `q` be an array of `m` elements q_j of the group Q_{x0} in **Leech lattice encoding**. Let g be an element of the group G_{x0} encoded in the array `g` of length `n` as in function `gen_leech2_op_word`.

The function computes the elements $q'_j = g^{-1}q_jg$ for $0 \leq j < m$ and stores q'_j in `a[j]`.

The function applies the same prefix of the word in the array `g` to all entries of `q`. It stops if the application of any prefix of the word in `g` to any element q_j fails; and it returns the length of the prefix of `a` that has been applied to all elements q_j. Hence the function returns `n` in case of success and a number `0 \leq k < n` if not all atoms of `q` could be applied to all entries of `q`.

The function succeeds on an individual entry q_j if and only if function `gen_leech2_op_word` succeeds on the same value.

uint32_t **`gen_leech2_op_word_leech2`**(uint32_t l, uint32_t *g, uint32_t n, uint32_t back)

Perform operation of G_{x0} on Leech lattice mod 2.

Let $g_0 \in G_{x0}$ be stored in the array referred by `g` as a word of generators of the subgroup G_{x0} of the monster. Here g_0 is given as a word of generators of length n in the array `g`. Each atom of the word g is encoded as defined in the header file `mmgroup_generators.h`. Put $g = g_0$ if `back == 0` and $g = g_0^{-1}$ otherwise.

The function returns the element $l \cdot g$ for a vector l in the Leech lattice mod 2.

Parameter l is encoded in **Leech lattice encoding**, igoring the sign. The function returns $l \cdot g$, in **Leech lattice encoding**, with the sign bit set to zero.

The function is optimized for speed. It returns garbage if any generator in the buffer `g` is not in G_{x0}.

int32_t **`gen_leech2_op_word_leech2_many`**(uint32_t *a, uint32_t m, uint32_t *g, uint32_t n, uint32_t back)

Perform operation of G_{x0} on elements ofLeech lattice mod 2.

Let `a` be an array of `m` elements l_j of the Leech lattice mod 2 in **Leech lattice encoding**. Let g_0 be an element of the group G_{x0} encoded in the array `g` of length `n` as in function `gen_leech2_op_word`. Put $g = g_0$ if `back == 0` and $g = g_0^{-1}$ otherwise.

The function replaces each element l_j in the array `m` by $l_j \cdot g$. Elements l_j are encoded in **Leech lattice encoding**, igoring the sign.

The function is optimized for speed. It returns zero in case of success. It stores garbage in the array `m` and returns a negative value if any generator in the buffer `g` is not in G_{x0}.

int32_t **`gen_leech2_op_word_matrix24`**(uint32_t *g, uint32_t n, uint32_t back, uint32_t *a)

Convert operation of group G_{x0} to 24 times 24 bit matrix.

Let $g_0 \in G_{x0}$ be stored in the array referred by `g` as a word of generators of the subgroup G_{x0} of the monster. Here g_0 is given as a word of generators of length n in the array `g`. Each atom of the word g is encoded as defined in the header file `mmgroup_generators.h`. Put $g = g_0$ if `back == 0` and $g = g_0^{-1}$ otherwise.

The function converts g to a 24×24 bit matrix a acting on the vectors of the Leech lattice mod 2 (encoded in **Leech lattice encoding**) by right multiplication. Such matrices form a natural representation of the Conway group Co_1.

The function returns 0 in case of success and a negative value in case of error.

void **`gen_leech2_op_mul_matrix24`**(uint32_t *a1, uint32_t n, uint32_t *a)

Multiply an n times 24 bit matrix with a 24 times 24 bit matrix.

The function multiplies the `n` times 24 bit matrix `a1` with the 24 times 24 bit matrix `a` and stores the result in matrix `a1`.

Thus array `a1` must have length `n`; and array `a1` must have length 24.

int32_t **`gen_leech2_map_std_subframe`**(uint32_t *g, uint32_t len_g, uint32_t *a)

Transform the standard subframe of Leech lattice mod 2

A **frame** in the Leech lattice Λ is a maximal set of pairwise orthogonal vectors of type 4. In $\Lambda/2\Lambda$ (which is the Leech lattice mod 2) a frame is mapped to a unique vector of type 4. The standard frame Ω in Λ consists of the type-8 vectors parallel to the basis vectors in Λ.

The subframe $S(F)$ of a frame F in Λ is the set $\{(u+v)/2 \mid u, v \in F, u \neq \pm v\}$. Any frame $S(F)$ in Λ contains 48 type-4 vectors, and its subframe contains $48 \cdot 46$ type-2 vectors.

The image of $S(F)$ in $\Lambda/2\Lambda$ spans a 12-dimensional maximal isotropic subpace $\langle S(F) \rangle$ of $\Lambda/2\Lambda$, and the type-2 vectors in $\langle S(F) \rangle$ are precisely images of $S(F)$.

Then for the standard frame Ω we have

$\langle S(\Omega) \rangle = \{\lambda_\delta, \lambda_\Omega + \lambda_\delta \mid \delta \in \mathcal{C}^*, \delta \, \mbox{even} \}$.

Here $\mathcal{C}^*$ is the Golay cocode, and and λ_c is the element of $\Lambda / 2 \Lambda$ corresponding to the Golay code or cocode element c.

Then $\langle S(\Omega) \rangle$ is spanned by λ_Ω and $\lambda_{\{0,j\}}, 1 \leq j < 24$. Here $\{i,j\}$ is the Golay cocode word corresponding to the sum of basis vectors i and j of $\mbox{GF}_2^{24}$.

The elements x_Ω and $x_{\{0,j\}}$ of the group Q_{x0} are preimages of λ_Ω and $\lambda_{\{0,j\}}$ under the natural homomorphism from Q_{x0} to $\Lambda / 2 \Lambda$. These elements of Q_{x0} play an important role in the representation 196883_x of the monster group. For computations in the subgroup G_{x0} of the monster we sometimes want to compute the images of these elements of Q_{x0} under conjugation by an element g of G_{x0}.

Let $g \in G_{x0}$ be stored in the array `g` of length `len_g` as a word of generators of the subgroup G_{x0} of the monster. Then this function computes the following 24 elements of Q_{x0}:

$x_\Omega^g, x_{\{0,1\}}^g, \ldots, x_{\{0,23\}}^g$.

The function stores these 24 elements in the array `a` (in that order) in **Leech lattice encoding**.

In case of success the function returns the number of entries of the word `g` being processed. It returns a negative value in case of failure.

C interface for file gen_leech_type.c

The functions in file `gen_leech_type.c` implement operations for detecting the type of a vector in the Leech lattice modulo 2.

We use the terminology defined in the document *The C interface of the mmgroup project*, section *Description of the mmgroup.generators extension*.

Functions

uint32_t **`gen_leech2_subtype`**(uint64_t v2)

Return subtype of vector in Leech lattice mod 2.

The function returns the subtype of the vector v_2 in the Leech lattice modulo 2 as a BCD-coded two-digit integer. v_2 must be given in Leech lattice encoding.

The subtype of a vector in the Leech lattice mod 2 is defined in **The mmgroup guide for developers**, section **Computations in the Leech lattice modulo 2**.

uint32_t **`gen_leech2_type`**(uint64_t v2)

Return type of a vector in the Leech lattice mod 2.

This function returns the type of the vector v_2 in the Leech lattice modulo 2. That type may be 0, 2, 3, or 4.

This function is a bit faster than function *`gen_leech2_subtype()`*.

uint32_t **`gen_leech2_type2`**(uint64_t v2)

Compute subtype if vector in Leech lattice mod 2 is of type 2.

This function returns the subtype of the vector v_2 in the Leech lattice modulo 2 if v_2 is of type 2 and 0 otherwise.

It is faster than function *`gen_leech2_type()`*.

uint32_t **gen_leech2_count_type2**(uint32_t *a, uint32_t n)

Count type-2 vectors in an affine subspace of the Leech lattice mod 2.

This function returns the number of type-2 vectors in an affine subspace V of the Leech lattice mod 2. Subspace V is defined by an array a of length n of bit vectors. If $a_1, \ldots, a_{n-1}$ are linear independent then V is given by:

$V = \{a_0 + \sum_{i=1}^{n-1} \lambda_i a_i \mid \lambda_i = 0, 1\}$.

int32_t **gen_leech2_start_type24**(uint32_t v)

Auxiliary function for function gen_leech2_reduce_type4.

The function returns the subtype of a vector v of type 2 in the Leech lattice modulo 2, provided that $v + \beta$ is of type 4. It returns 0 in the special case $v = \beta + \Omega$ and a negative value if v is not of type 2 or $v + \beta$ is not of type 4.

It is used for rotating a type-4 vector v which is orthogonal to β (in the real Leech lattice) into the $v + \Omega$. That rotation will fix the special short vector β.

int32_t **gen_leech2_start_type4**(uint32_t v)

Auxiliary function for function gen_leech2_reduce_type4.

The function returns the subtype of a vector v of type 4 in the Leech lattice modulo 2. Parameter v must a vector of type 4 in Leech lattice encoding. The function returns the subtype of v that will be used for reduction in function `gen_leech2_reduce_type4`.

This function takes care of the special vectors Ω and β the Leech lattice modulo 2.

It is used for rotating a type-4 vector v into Ω. If this is possible, that rotation will fix the special short vector β.

Therefore the function returns 0 in case $v = \Omega$. It returns the subtype of $v + \beta$ if β and $v + \beta$ are of type 2 and orthogonal in the real Leech lattice.

The function returns a negative value if v is not of type 4.

C interface for file gen_leech3.c

The functions in file `gen_leech3.c` implement operations on the vectors of the Leech lattice modulo 3 and on the subgroup Q_{x0}. We use the terminology defined in the document *The C interface of the mmgroup project*, section *Description of the mmgroup.generators extension*.

Functions

uint32_t **gen_leech3_scalprod**(uint64_t v3_1, uint64_t v3_2)

Scalar product of two vectors in the Leech lattice mod 3.

The function returns the scalar product of the vectors $v_{3,1}, v_{3,2}$. The parameters are given in Leech lattice mod 3 encoding. The result is between 0 and 2.

uint64_t **gen_leech3_add**(uint64_t v3_1, uint64_t v3_2)

Add two vectors in the Leech lattice mod 3.

The function returns the sum of the vectors $v_{3,1}, v_{3,2}$. The parameters and the result are given in Leech lattice mod 3 encoding.

uint64_t **gen_leech3_neg**(uint64_t v3)

Negate a vector in the Leech lattice mod 3.

The function returns the negated vector v_3. The parameter and the result are given in Leech lattice mod 3 encoding.

uint64_t **gen_leech2to3_short**(uint64_t v2)

Map short vector from $\Lambda/2\Lambda$ to $\Lambda/3\Lambda$.

Here parameter v_2 is a short vector (i.e. a vector of type 2) in $\Lambda/2\Lambda$ in Leech lattice encoding.

The function returns a short vector in $\Lambda/3\Lambda$ corresponding to v_2 in Leech lattice mod3 encoding.

The result is unique upto sign only. The function returns 0 if v_2 is not short.

uint64_t **gen_leech2to3_abs**(uint64_t v2)

Map vector from $\Lambda/2\Lambda$ to $\Lambda/3\Lambda$.

Here parameter v_2 is a short vector of type 2 or 3. in $\Lambda/2\Lambda$ in Leech lattice encoding.

The function returns a vector in $\Lambda/3\Lambda$ corresponding to v_2 in Leech lattice mod3 encoding.

The result is unique upto sign only. The function returns 0 if v_2 is not of type 2 or 3.

uint64_t **gen_leech3to2_short**(uint64_t v3)

Map short vector from $\Lambda/3\Lambda$ to $\Lambda/2\Lambda$.

Here parmeter v_3 is a short vector (i.e. a vector of type 2) in $\Lambda/3\Lambda$ in Leech lattice mod 3 encoding.

The function returns a short vector in $\Lambda/2\Lambda$ corresponding to v_3 in Leech lattice encoding.

The result is unique. The function returns 0 if v_3 is not short. This function is an inverse of function `gen_leech2to3_short`.

uint64_t **gen_leech3to2**(uint64_t v3)

Map vector from $\Lambda/3\Lambda$ to $\Lambda/2\Lambda$.

Here parameter v_3 is a vector in $\Lambda/3\Lambda$ in Leech lattice mod 3 encoding.

If a shortest preimage v of v_3 in Λ if of type t with $t \leq 4$ then the function computes the (unique) vector v_2 in $\Lambda/2\Lambda$ that has the same preimage v in Λ. Otherwise the function fails.

In case of success the function returns $2^{24} \cdot t + v_2$, with v_2 given in Leech lattice encoding. The function returns `uint64_t(-1)` in case of failure.

uint64_t **gen_leech3to2_type4**(uint64_t v3)

Map type-4 vector from $\Lambda/3\Lambda$ to $\Lambda/2\Lambda$.

Here parameter v_3 must be a type-4 vector in $\Lambda/3\Lambda$ in Leech lattice mod 3 encoding.

The function returns a type-4 vector in $\Lambda/2\Lambda$ corresponding to v_3 in Leech lattice encoding.

The result is unique. The function returns 0 if v_3 is not of type 4.

uint64_t **gen_leech3_op_xi**(uint64_t v3, uint32_t e)

Special case of function `gen_leech3_op_vector_word`

For internal purposes only. This is equivalent to `gen_leech3_op_vector_word(v3, g)`, where `g` encodes the element ξ^e of G_{x1}.

Parameter v_3 and the result are given Leech lattice mod 3 encoding.

uint64_t **gen_leech3_op_vector_word**(uint64_t v3, uint32_t *pg, uint32_t n)

Operation of G_{x1} on the Leech lattice mod 3.

The function returns the element $v_3 g$ for $v_3 \in \Lambda/3\Lambda$ and $g \in G_{x0}$. Here g is given as a word of genenators of length n in the array `pg`. Each atom of the word g is encoded as defined in the header file `mmgroup_generators.h`.

Parameter v_3 and the result are given Leech lattice mod 3 encoding.

uint64_t **gen_leech3_op_vector_atom**(uint64_t v3, uint32_t g)

Atomic operation of G_{x1} on the Leech lattice mod 3.

Equivalent to `gen_leech3_op_vector_word(v3, &g, 1)`.

Parameter v_3 and the result are given Leech lattice mod 3 encoding.

C interface for file gen_leech_reduce.c

The functions in file `gen_leech_reduce.c` implement the transformation of vectors of the Leech lattice modulo 2 to a standard form by applying an element of the subgroup G_{x0} of the monster.

We use the terminology defined in the document *The C interface of the mmgroup project*, section *Description of the mmgroup.generators extension*.

Functions

int32_t **gen_leech2_reduce_type2**(uint32_t v, uint32_t *pg_out)

Map short vector in Leech lattice to standard vector.

Let $v \in \Lambda/2\Lambda$ of type 2 be given by parameter `v` in Leech lattice encoding. Then the function constructs a $g \in G_{x0}$ that maps v to the standard short vector v_0. Here v_0 is the short vector the Leech lattice propotional to $e_2 - e_3$, where e_i is the i-th basis vector of $\{0, 1\}^{24}$.

The element g is returned as a word in the generators of G_{x0} of length $n \leq 6$. Each atom of the word g is encoded as defined in the header file `mmgroup_generators.h`.

The function stores g as a word of generators in the array `pg_out` and returns the length n of that word. It returns a negative number in case of failure, e.g. if v is not of type 2.

Caution:

An input vector allows several outputs. Changing the implementation of this function such that the same input leads to a different output destroys the interoperability between different versions of the project!!

int32_t **gen_leech2_reduce_type2_ortho**(uint32_t v, uint32_t *pg_out)

Map (orthogonal) short vector in Leech lattice to standard vector.

Let $v \in \Lambda/2\Lambda$ of type 2 be given by parameter `v` in Leech lattice encoding.

In the real Leech lattice, (the origin of) the vector v must be orthogonal to the standard short vector v_0. Here v_0 is the short vector in the Leech lattice propotional to $e_2 - e_3$, where e_i is the i-th basis vector of $\{0, 1\}^{24}$.

Let v_1 be the short vector in the Leech lattice proportional to $e_2 + e_3$. Then the function constructs a $g \in G_{x0}$ that maps v to v_1 and fixes v_0.

The element g is returned as a word in the generators of G_{x0} of length $n \leq 6$. Each atom of the word g is encoded as defined in the header file `mmgroup_generators.h`.

The function stores g as a word of generators in the array `pg_out` and returns the length n of that word. It returns a negative number in case of failure, e.g. if v is not of type 2, or not orthogonal to v_0 in the real Leech lattice.

Caution:

An input vector allow several outputs. Changing the implementation of this function such that the same input leads to a different output destroys the interoperability between different versions of the project!!

int32_t **gen_leech2_reduce_type4**(uint32_t v, uint32_t *pg_out)

Map a frame in the Leech lattice to the standard frame.

A frame in the Leech lattice Λ is a maximal set of type-4 vectors which are equal modulo 2Λ. A frame is equivalent to a type-4 vector in $\Lambda/2\Lambda$.

Let $v \in \Lambda/2\Lambda$ of type 4 be given by parameter `v` in Leech lattice encoding. Then the function constructs a $g \in G_{x0}$ that maps v to the standard frame Ω. The standard frame Ω consists of the type-4 vectors parallel to the coordinate axes.

The element g is returned as a word in the generators of G_{x0} of length $n \leq 6$. Each atom of the word g is encoded as defined in the header file `mmgroup_generators.h`. Let H be the stabilizer of Ω. We choose a representative g in the coset gH such that the inverse g^{-1} is a short as possible.

The function stores g as a word of generators in the array `pg_out` and returns the length n of that word. It returns a negative number in case of failure, e.g. if v is not of type 4.

The function uses the method described in the `The mmgroup guide for developers`, section `Computations in the Leech lattice modulo 2`.

We make one additional assertion.

Let v_0 be the standard short vector in the Leech lattice proportional to $e_2 - e_3$, where e_i is the i-th basis vector of $\{0,1\}^{24}$. If $v_0 + v$ is of type 2 then this function computes the same result as function `gen_leech2_reduce_type2_ortho` applied to the vector $v_0 + v$. In this case the result of this function centralizes v_0. This convention greatly simplifies computations in the baby monster group.

Caution:

An input vector allow several outputs. Changing the implementation of this function such that the same input leads to a different output destroys the interoperability between different versions of the project!!

uint32_t **gen_leech2_type_selftest**(uint32_t start, uint32_t n, uint32_t *result)

Test functions `gen_leech2_subtype` and `gen_leech2_reduce_type4`

Function `gen_leech2_subtype` may tested as follows:

We compute the subtypes of all 2^{24} vectors $v \in \Lambda/2\Lambda$ and we count the obtained subtypes in an array `result` of length 0x50. The sizes of the orbits of each subtype are known form [Iva99], so a high-level test routine may check the array `result`.

During that process we may also test function `gen_leech2_reduce_type4`. Whenever vector of type-4 vector v occurs, we compute a `g \in G_{x0}` that maps v to the standard frame using function `gen_leech2_reduce_type4`; and we check the correctness of that mapping using function `gen_leech2_op_word`. We return the number of successful such operations. This number must be equal to the number of type-4 vectors in $\Lambda/2\Lambda$.

Since this test takes a long time, a high-level function might want to distribute it over several processes. So this function acccumulates the test results for the vectors v with start $\leq v <$ start $+$ n only.

int32_t **gen_leech2_pair_type**(uint64_t v1, uint64_t v2)

Isomorphsim type of pair of vectors in Leech lattice mod 2.

Let v_1 and v_2 be two vectors in the Leech lattice mod 2 and put $v_3 = v_1 + v_2$. Let $t_i, i = 1,2,3$ be the type of v_i. Then the triple (t_1, t_2, t_3) depends on the isomorphism type of the pair (v_1, v_2).

If one or three of the vectors (v_1, v_2, v_3) are of type 4 then there may be several isomorphism types of pairs (v_1, v_2) with the same triple (t_1, t_2, t_3). In this case we define a subtype s as follows. We map one of the vectors v_i of type 4 to the standard frame Ω. Let w be the image of any of the vectors $v_j, j \neq i$ under any such mapping. Then we compute the subtype of w with function `gen_leech2_subtype`, and we let s be the last hexadecimal digit of the computed subtype. The result s does not depend on the choices made in the description given above.

After reordering t_1, t_2, t_3, the possibilties for tuples (t_1, t_2, t_3, s) with a relevant component s are (4,2,2,0), (4,3,3,4), (4,3,3,6), (4,4,4,0), (4,4,4,4), (4,4,4,6).

If none of the vectors (v_1, v_2, v_3) is of type 4 then all the shortest corresponding vectors in the Leech lattice are defined up to sign. Then these vectors may span a space of dimension 2 or 3; and we put $s = 1$ if that dimension is equal to 3, and $s = 0$ otherwise. The case $s = 1$ may occur if and only if at least two of the vectors (v_1, v_2, v_3) are of type 3 and none is of type 4.

In all other cases there is at most one such isomorphism type; and we put $s = 0$.

The function takes v_1 and v_2 as parameters `v1` and `v2` in Leech lattice encding. It returns

$2^{12} \cdot t_1 + 2^8 \cdot t_2 + 2^4 \cdot t_3 + s$.

We sketch a proof that the returned result actually describes the isomorphism type of the pair (v_1, v_2). If one of the vectors (v_1, v_2, v_3) is of type 4 then may map that vector to the standard frame Ω and check the possible subtypes of the other vectors. These subtypes are described in subsection **Computations in the Leech lattice modulo 2** of **The guide for developers**, or in [Iva99], Section 4.4 ff.

Otherwise the shortest preimages v_1', v_2', v_3' of v_1, v_2, v_3 in the Leech lattice are defined up to sign. The cases where these preimages span a subspace of dimension 2 of the Leech lattice are well known, see e.g. [CS99] or [CCN+85]. If v_1', v_2', v_3' span a 3-dimensional space, then it is easy to see that at least two of these vectors must be of type 3, and that all vectors $(v_1' \pm v_2' \pm v_3')/2$ (with all possible combinations of signs) are in the Leech lattice and of type at most 3. The 3-dimensional lattices satifying these properties can be shown to be S-lattices, as defined in [CCN+85]. Note that the S-lattices are also known.

Warning: This function has not yet been tested!

C interface for file gen_leech_reduce_n.c

Given a vector v in the extraspecial group Q_{x0}, the functions in this module compute a transformation from v to a standard representative of the class v^N.

Here Q_{x0} and $N = N_{x0}$ are as in the project documentation.

We use the terminology defined in the document *The C interface of the mmgroup project*, section *Description of the mmgroup.generators extension*.

Functions

int32_t **gen_leech2_reduce_n**(uint32_t v, uint32_t *pg_out)

Reduce Leech lattice vector modulo the group N_x0.

Given a vector v in the extraspecial group Q_{x0}, we want to find a standard representative of the class v^N, where Q_{x0} and $N = N_{x0}$ are as in the project documentation. Here v is given in parameter `v` in **Leech lattice encoding**.

This function computes such a representative $v_1 \in Q_{x0}$. It returns an element r of N_{x0} with $v_1 = v^r$ as a word of length three in the generators of N_{x0}. That word is stored in the buffer referred by `pg_out` as an array of 3 integers of type `uint32_t` as described in section **The monster group** in the API reference.

The user may call function `gen_leech2_op_word` in file `gen_leech.c` for computing $v_1 = v^r$. There is a faster alternative for computing v_1 described in the documetnation of function `gen_leech2_reduce_n`.

The representative v_1 depends on the **subtype** of v only, with one exception listed below. Here the subtype can be also computed by function `gen_leech2_subtype`in file `gen_leech.c`; it is explained in **The mmgroup guide for developers** in section **Computations in the Leech lattice modulo 2**.

If the subtype of v is `00`then we have $v_1 = v$ for the two vectors $v = x_{\pm 1}$ of that subtype.

We compute a mapping $r = x_\pi y_e x_f$, where r is an automorphism of the Parker loop, e is in the Parker loop, and f is either in the Parker loop or in the Golay cocode.

The result r is in the even subgroup of N_{x0} except in case $v = x_{-\Omega}$, where this is not possible.

See section **Orbits of the group N_x0 in the group Q_x0** in **The mmgroup guide for developers** for our choice of the representative of the N_{x0} orbits.

int32_t **gen_leech2_reduce_n_rep**(uint32_t subtype)

Compute standard representative of Q_x0 modulo the group N_x0.

Given a subtype `subtype` in the Leech lattice modulo 2, the function returns a representative of a nonzero orbit of Q_{x0} under the action of N_{x0}.

See the table in section **Orbits of the group N_x0 in the group Q_x0** in **The mmgroup guide for developers** for our choice of such a representative.

C interface for file gen_leech_reduce_22.c

Let $\Lambda/2\Lambda$ be the Leech lattice modulo 2. Let $v_2, v_3 \in \Lambda/2\Lambda$ with type$(v_2) = 2$ and type$(v_3) =$ type$(v_2 + v_3) = t$ for $2 \leq t \leq 3$. We want to compute the set of all vectors $v_4 \in \Lambda/2\Lambda$ of type 4 with type$(v_2 + v_4) =$ type$(v_3 + v_4) =$ type$(v_2 + v_3 + v_4) = 2$. There are 891 or 100 such vectors in case $t = 2$ or $t = 3$, respectively. Function `gen_leech2_u4_2xx` in this module computes the list of all such vectors.

We motivate this (rather complicated) computation as follows. The main algorithm for the reduction of an element of the Monster in Version v1.0.0 the **mmgroup** package works in the representation ρ_{15} of the monster, with coefficients taken mod 15. In future versions it would be faster to work in representation ρ_3 instead. For the reduction algorithm we have to compute certain sets $U_4(v)$ for axis v in the representation of the Monster as defined in [Sey22]. This computation is considerably more difficult for $v \in \rho_3$ than for $v \in \rho_{15}$. It turns out that for $t = 2, 3$ the sets computed above are just the sets $U_4(v)$ when v is in the orbit called '6A' or '10A' in [Sey22]. Given an axis $v \in \rho_3$ in one of these orbits, we can effectively compute suitable vectors v_2, v_3 as discussed above.

For that computation we need a function that maps vectors v_2, v_3 to fixed vectors $w_2, w_{3t} \in \Lambda/2\Lambda$. Function `gen_leech2_reduce_2xx` essentially performs this task; see the documentation of that function for details.

The group fixing v_2 and v_3 is the unitary group $\mathrm{PSU}_6(2)$ in case $t = 2$, and the Higman-Sims group in case $t = 3$, see [CS99], Ch. 10.3.4. A detailed discussion of the case $t = 3$ is given in [CS99], Ch. 10.3.5.

We use the terminology defined in the document *The C interface of the mmgroup project*, Section *Description of the mmgroup.generators extension*. Vectors in $\Lambda/2\Lambda$ are given in *Leech lattice encoding*; the sign bit is ignored on input and set to zero on output.

Functions

int32_t **gen_leech2_n_type_22**(uint32_t n)

Generate certain type-2 vectors in the Leech lattice.

The function maps the integers $0 \leq n < 4600$ to the 4600 type-2 vectors v in the Leech lattice mod 2, such that that $v + w$ is also of type 2 in the Leech lattice mod 2. Here w is the standard type-2 vector $(0, 0, 4, -4, 0, \ldots, 0)$. Such a vector v is returned in *Leech lattice encoding*.

The function returns a negative value in case of failure; e.g. if $n \geq 4600$.

int32_t **gen_leech2_find_v4_2xx**(uint32_t v2, uint32_t v3, uint64_t *seed)

Support for reducing a certain pair of vectors in the Leech lattice.

Let `v2` be a vector of type 2 in the Leech lattice mod 2, and let `v3` be a vector of type t in the Leech lattice mod 2, with 2 <= t <= 3, such that `v2 + v3` is also of type t. Here `v2` and `v3` must be given in *Leech lattice encoding*.

The function computes a vector `v4` of type 4 such that `v2 + v4, v3 + v4`, and `v2 + v3 + v4` are all of type 2. Vector `v4` can be used in function `gen_leech2_find_v4_2xx` for reducing the pair `(v2, v3)` to a standard pair of vectors.

The function returns the value `t * 0x1000000 + v4` in case of succcess, with t as above. Parameter `seed` is a seed for a random generator as described in module `gen_random.c`. The function returns a negative value in case of failure.

The function tries to find random vectors `vx` of type 2 in the Leech lattice mod 2 such that both, `vx + v2`, and `vx + v3` are of type 2. Then the vector `v4 = v2 + v3 + vx` satifies the required conditions.

For finding `vx` we transform `v2` to the standard type-2 vector with a transformation `tf`. Then we generate random type-2 vectors with function `gen_leech2_n_type_22`. These random type-2 vectors are interpreted as transformed vectors `tf(vx)`; and we will check the required conditions on the transformed vectors `tf(v2)`, `tf(v3)`, and `tf(vx)`.

int32_t **`gen_leech2_reduce_2xx`**(uint32_t v2, uint32_t v3, uint32_t v4, uint32_t *g)

Reduce a certain pair of vectors in the Leech lattice.

Let `v2` be a vector of type 2 in the Leech lattice mod 2, and let `v3` be a vector of type t in the Leech lattice mod 2, with 2 <= t <= 3, such that `v2 + v3` is also of type t. Here `v2` and `v3` must be given in *Leech lattice encoding*.

The function computes an element `g` of the Conway group Co_1 such that `v2 * g = w2` and `v3 * g = w3` with `w2 = (0, 0, 4, -4, 0, ..., 0)`. Here we have `w3 = (0, 4, -4, 0, ..., 0)` if `t == 2` and `w3 = (1, 1, 5, 1, ..., 1)` if `t == 3`.

The function stores `g` in the array `g` as a word of length `k` of generators of the subgroup `G_x0` of the Monster. These generators are encoded as described in file `mmgroup_generators.h`. We have `k <= 8`; so the array `g` must have size at least 8.

The function returns the value `k >= 0` in case of succcess, and a negative value in case of failure.

Parameter `v4` must be a type-4 vector satisfying certain conditions as described in de documentation of function `gen_leech2_find_v4_2xx`. That function computes a suitable random vector `v4`.

Internally, we use function `gen_leech2_reduce_type4` to find an element `g_0` of Co_1 that maps `v4` to the standard type-4 vector `Omega`. To finish up, we just have to calculate an element `g_1` of the subgroup $2^{11}.M_{24}$ of Co_1 such that `g = g_0 * g_1` satisfies the required conditions.

int32_t **`gen_leech2_map_2xx`**(uint32_t *g, uint32_t n, uint32_t t, uint32_t *a)

Auxiliary function for function `gen_leech2_u4_2xx`

Let `v2`, `v3` be as in function `gen_leech2_reduce_2xx`. Let `g` be an element of the group `G_x0` of length `n` that maps `v2` and `v3` to `w2` and `w3`, with `w2`, `w3` as in function `gen_leech2_reduce_2xx`. Here `g` should have been computed by that function.

The function computes all vectors `v4` of type 4 in the Leech lattice mod 2 such that `v2 + v4, v3 + v4`, and `v2 + v3 + v4` are of type 2. It stores the list of these vectors in the array `a` and returns the length of that list.

Array `a` must have length 891 in case `t == 2` and length 100 in case `t == 3`. Other values of `t` are illegal.

int32_t **`gen_leech2_u4_2xx`**(uint32_t v2, uint32_t v3, uint64_t *seed, uint32_t *a)

Compute a certain set of type-4 vectors in the Leech lattice mod 2.

Let `v2` be a vector of type 2 in the Leech lattice mod 2, and let `v3` be a vector of type t in the Leech lattice mod 2, with 2 <= t <= 3, such that `v2 + v3` is also of type t. Here `v2` and `v3` must be given in *Leech lattice encoding*.

The function computes all vectors `v4` of type 4 in the Leech lattice mod 2 such that `v2 + v4, v3 + v4`, and `v2 + v3 + v4` are of type 2. There are 891 such vectors in case `t = 2` and 100 such vectors in case `t = 3`. The list all these vectors is stored in the array `a` in *Leech lattice encoding* in an undefined order. Thus array `a` must have length at least 891.

The function uses a random generator. Parameter `seed` is a seed for a random generator as described in module `gen_random.c`.

The function returns the length of the array `a` in case of succcess, and a negative value in case of failure.

C interface for file gen_union_find.c

The functions in file `gen_union_find.c` implement a union-find algorithm. This may be used e.g. for computing the orbits of the Leech lattice mod 2 under the action of a group.

Functions

int32_t **`gen_ufind_init`**(uint32_t *table, uint32_t length)

Initialize table for union-find algorithm.

The function initializes an array referred by parameter `table` for performing a union-find algorithm on the set `S(length)` of integers `i` with `0 <= i < length`. Array `table` must have size `length`. Here `0 <= length <= 0x40000000`must hold.

Then `table` will store a partition of the set `S(length)` in an internal format. Initially, that partition consists of singletons. One may use function `gen_ufind_union` for joining two sets of that partition. Any set in the partition is represented by an element of that set. The rules for computing a representative of a set are not disclosed to the user. You may use function `gen_ufind_find` for finding the representative of a set. After calling function `gen_ufind_find_all_min` each set is represented by its smallest element. The representative of a set may change after calling function `gen_ufind_union`.

The function returns 0 in case of success, and a negative value in case of failure (e.g. if `length` is too big.)

int32_t **`gen_ufind_find`**(uint32_t *table, uint32_t length, uint32_t n)

The find function for the union-find algorithm.

Let `S(length)` be the set of integers `i` with `0 <= i < length`; and let a partition of `S(length)` be stored in the array `table` (of size `length`) as described in function `gen_ufind_init`.

The function returns the representative of the set containing the integer `n`. It returns a negative number in case of failure, e.g. if `n >= length`.

void **`gen_ufind_union`**(uint32_t *table, uint32_t length, uint32_t n1, uint32_t n2)

The union function for the union-find algorithm.

Let `S(length)` be the set of integers `i` with `0 <= i < length`; and let a partition of `S(length)` be stored in the array `table` (of size `length`) as described in function `gen_ufind_init`.

The function joins the sets in the partition containg the elements `n1` and `n2`.

int32_t **`gen_ufind_find_all_min`**(uint32_t *table, uint32_t length)

Choose smallest representatives for sets in union-find algorithm.

Let `S(length)` be the set of integers `i` with `0 <= i < length`; and let a partition of `S(length)` be stored in the array `table` (of size `length`) as described in function `gen_ufind_init`.

The function chooses the smallest element of a set in a partition as the representative of the set. Thus a subsequent call to function `gen_ufind_find` with parameter `n` returns the smallest element of the set containg `n`.

The function returns the number of the sets in the partition. A negative return value indicates a error the array `table`.

int32_t **gen_ufind_partition**(uint32_t *table, uint32_t length, uint32_t *ind, uint32_t len_ind)

Output the partition computed by the union-find algorithm.

Let `S(length)` be the set of integers `i` with `0 <= i < length`; and let a partition of `S(length)` be stored in the array `table` (of size `length`) as described in function `gen_ufind_init`.

We assume that the partition stored in the array `table` contains `n_sets` sets. Then we overwrite the array `table` with a list of lists, where each list corresponds to a set in the partition. We write some index infomation for interpreting these lists into the array `ind` of length `len_ind`.

Array `table` will contain the `length` elements of the set `S(length)` in such a way that for `0 <= i < n_sets` the `i`-th set in the partition is equal to the set of integers given by `table[ind[i]], ..., table[ind[i+1] - 1]`.

So the array `ind` must have size at least `n_sets + 1`; i.e. we must have `len_ind` > `n_sets`. Function `gen_ufind_find_all_min` must be called beford calling this function. Note that function `gen_ufind_find_all_min` returns `n_sets` in case of success.

The entries `table[ind[i]], ..., table[ind[i+1] - 1]` corresponding to set in the partition are sorted. The sets of the partition are sorted by their smallest elements.

The function returns `n_set` in case of success and a negative value in case of failure.

void **gen_ufind_union_affine**(uint32_t *table, uint32_t length, uint32_t *m, uint32_t len_m, uint32_t v)

Perform affine union operations in the union-find algorithm.

Let `S(length)` be the set of integers `i` with `0 <= i < length`; and let a partition of `S(length)` be stored in the array `table` (of size `length`) as described in function `gen_ufind_init`.

In this function the entries of `S(length)` are interpreted as bit vectors. The function joins the set containing `i` with the set containg `i * M + v` for all `i`. Here `M` the bit matrix over GF(2) referred by `m` with `len_m` rows, and `v` a bit vector. All bit vector arithmetic is done over GF(2).

int32_t **gen_ufind_union_leech2**(uint32_t *table, uint32_t *g, uint32_t len_g)

Join orbits of the Leech lattice mod 2.

The Conway group Co_1 has a natural action on on $\Lambda/2\Lambda$, i.e. on the Leech lattice mod 2. The subgroup G_{x0} of the Monster (of structure $2^{1+24}.\mathrm{Co}_1$) has the same action on on $\Lambda/2\Lambda$.

In this function we assume that the array `table` contains a partition of the set $\{i \mid 0 \leq i < 2^{24}\}$ as described in function `gen_ufind_init`. In that function the array `table` should have been initialized with a length of 2^{24}.

Here each integer i is interpreted as an element of $\Lambda/2\Lambda$ in *Leech lattice encoding*, as described in the document *The C interface of the mmgroup project*, section *Description of the mmgroup.generators extension*.

Let $g \in G_{x0}$ be stored in the array referred by `g` as a word of generators of the subgroup G_{x0} of the monster. Here g is given as a word of generators of length `len_g` in the array `g`. Each atom of the word g is encoded as defined in the header file `mmgroup_generators.h`.

Then the function joins the set containing i with the set containing $i \cdot g$ for all i, using the union-find algorithm.

The function returns 0 in case of success and a negative value in case of failure (e.g. if g is not in G_{x0}).

3.3.8 C functions for the generator ξ of the monster group

Reference implementation for module `gen_xi_functions.c`

The methods in class `GenXi` in this module are a reference implementations of the functions in module `gen_xi_functions.c`. These functions deal with the operation of the generator ξ of the monster defined in [Sey20], Section 9, on a subgroup Q_{x0}, and also with the monomial part of that operation on the 196884-dimensional representation ρ of the monster.

Generator ξ is an element of the subgroup G_{x0} of the monster. It operates on the normal subgroup Q_{x0} of G_{x0} of structure 2^{1+24} by conjugation as described in cite:*Seysen20*, Lemma 9.5. Method `gen_xi_op_xi` of class `GenXi` implements the operation of the generators ξ^e on the group Q_{x0}. Here the elements of Q_{x0} are given in *Leech lattice encoding*, as described in *The C interface of the mmgroup project*, Section *Description of the mmgroup.generators extension*.

Representation ρ has a subspace 98280_x of dimension 98280 on which G_{x0} operates monomially in the same way as it operates on the short elements of Q_{x0}. Here the basis vectors of 98280_x (together with their opposite vectors) can be identified with the short elements of Q_{x0}.

For a fast implementation of the operation of ξ^e on 98280_x we precompute tables as described in the *mmgroup guide for developers*, Section *Some mathematical aspects of the implementation*, Subsection *Implementing generators of the Monster group*, Subsubsection *Monomial operation of the generators xi^e*. We adopt the terminology from that subsection.

In that subsection the basis vectors of 98280_x are subdivided into five boxes, so that ξ acts as a permutation on these boxes. Within each of these boxes the basis vectors of 98280_x are numbered in a natural way as described in that subsection. Operator ξ permutes the five boxes 1,…,5 as follows:

- 1 -> 1, 2 -> 2, 3 -> 4 -> 5 -> 3 .

For each short vector in Q_{x0} we encode the number of the box containing the vector, the position of the vector inside the box, and and the sign of the vector in the lowest 19 bits of a 32-bit integer as follows:

- Bit 18…16: number of the box, running from 1 to 5.
- Bit 15: sign bit
- Bit 14…0: Entry inside the box

We call this encoding the *Short vector encoding* of a short vector in Q_{x0}.

Methods `gen_xi_short_to_leech` and `gen_xi_leech_to_short` in class `GenXi` convert elements of Q_{x0} from *Leech lattice encoding* to *Short vector encoding* and vice versa. Method `gen_xi_op_xi_short` uses these two methods together with method `gen_xi_op_xi` in order to compute the operation of ξ^e on a vector in 98280_x encoded in *Short vector encoding*.

Using method `gen_xi_op_xi_short` we may easily compute tables for the operation of ξ and ξ^2 on a specific box. Method `gen_xi_make_table` computes an array containing the lower 16 bits of the images of the entries of a box under the operation ξ^e. Bits 18…16 of these images can be reconstructed from the permutation of the boxes given above.

C interface for file gen_xi_functions.c

The functions in file `gen_xi_function.c` correspond to the python functions in class `mmgroup.dev.generators.gen_xi_ref.GenXi`. They are used for computing the tables required for implementing the operation of generator ξ on the representation ρ_p of the Monster.

Functions

uint32_t **`gen_xi_g_gray`**(uint32_t v)

Compute function γ on a Golay code element.

Here parameter v is encoded in `gcode` representation, and the return value $\gamma(v)$ is encoded in `cocode` representation. These representations are defined in the *Description of the mmgroup.mat24 extension*.

uint32_t **`gen_xi_w2_gray`**(uint32_t v)

Compute function w_2 on a Golay code element.

Here parameter v is encoded in `gcode` representation as defined in the *Description of the mmgroup.mat24 extension*. The function returns $w_2(v)$, which may be 0 or 1.

uint32_t **`gen_xi_g_cocode`**(uint32_t c)

A kind of an inverse of function γ.

Given a cocode element c in `cocode` representation, the function returns the unique grey Golay code vector v such that $\gamma(v)$ is equal to the grey part of c. v is returned in `gcode` representation, see the *Description of the mmgroup.mat24 extension*.

uint32_t **`gen_xi_w2_cocode`**(uint32_t v)

Compute function w_2 on a Golay cocode element.

Here parameter v is encoded in `cocode` representation as defined in the *Description of the mmgroup.mat24 extension*. The function returns $w_2(v)$, which may be 0 or 1.

uint32_t **`gen_xi_op_xi`**(uint32_t x, uint32_t e)

Conjugate an element of the group Q_{x0} with ξ^e.

The function returns $\xi^{-e} x \xi^e$ for an element x of the group Q_{x0}. Input x and the return value are encoded in *Leech lattice encoding* as described above.

uint32_t **`gen_xi_op_xi_nosign`**(uint32_t x, uint32_t e)

Similar to function `gen_xi_op_xi`, ingoring sign.

The function returns $\xi^{-e} x \xi^e$ for an element x of the group Q_{x0}. Input x and the return value are encoded in *Leech lattice encoding* as described above. This function computes the result up to sign only.

uint32_t **`gen_xi_leech_to_short`**(uint32_t x)

Convert the encoding of an element of the group Q_{x0}.

The function converts an element x of the group Q_{x0} from *Leech lattice encoding* to *Short vector encoding* and returns the converted element.

uint32_t **`gen_xi_short_to_leech`**(uint32_t x)

Convert the encoding of an element of the group Q_{x0}.

The function converts an element x of the group Q_{x0} from *Short vector encoding* to *Leech lattice encoding* and returns the converted element.

An invalid value x is converted to 0.

uint32_t **gen_xi_op_xi_short**(uint32_t x, uint32_t u)

Conjugate an element of the group Q_{x0} with ξ^e.

The function returns $\xi^{-e} x \xi^e$ for an element x of the group Q_{x0}. In contrast to function gen_xi_op_xi, the input x and the return value are encoded in *short vector encoding* as described above.

Any invalid element x is converted to x.

uint32_t **gen_xi_make_table**(uint32_t b, uint32_t e, uint16_t *ptab)

Create table for operation of ξ^e,.

The function creates a table `ptab` such that `ptab[i]` encodes $\xi^{-e} x \xi^e$, where $x = 2^{16} \cdot b + i$ encodes a short vector in Q_{x0} in *Short vector encoding*. The length of the gererated table is $l(b) = 2496, 23040, 24576, 32768, 32768$ for $b = 1, 2, 3, 4, 5$.

Here `ptab[i]` contains the lower 16 bits of the table entry only, with bit 15 equal to the sign bit. The upper 3 bits of `ptab[i]` depend on `b` only; they can be deduced from the permutation of the boxes `b = 1,...,5` by ξ described above.

Not all table indices `i` correspond to short vectors. We put `p[i] = i` for all invalid indices `i`.

void **gen_xi_invert_table**(uint16_t *ptab, uint32_t len, uint32_t ncols, uint16_t *pres, uint32_t len_res)

Invert a table created by function `gen_xi_make_table`

The function computes the inverse table of a table `ptab` of length `len` created by function `gen_xi_make_table` and stores the inverse table in the buffer referred by `pres`.

Parameter `len_res` is the length of the returned table. Parameter `ncols` must be 24 or 32. This means that only entries `ptab[32*i + j]` with `0 <= j < ncols` are valid.

The details of the inversion of such a table are rather tricky; they are coded in class `mmgroup.dev.mm_basics.mm_tables_xi.Pre_MM_TablesXi`. The purpose of this function is to speed up the calculations in that class.

void **gen_xi_split_table**(uint16_t *ptab, uint32_t len, uint32_t mod, uint32_t *psign)

Split a table created by function `gen_xi_invert_table`

The function splits the sign bit from a table `ptab` of length `len` created by function `gen_xi_invert_table` and replaces `ptab[i]` by `(ptab[i] & 0x7fff) % mod`. The sign bit `ptab[32*i+j] >> 15` is stored in bit `j` of entry `psign[i]`.

The purpose of this function is to speed up the computations in class `mmgroup.dev.mm_basics.mm_tables_xi.Pre_MM_TablesXi`.

3.3.9 C functions implementing a random generator

The python extension `mmgroup.generators` provides a random generator. Its main purpose is the very fast generation of a vector of integers modulo p for the representation ρ_p of the monster group for small numbers p.

The internal operation of the random generator is described in file `gen_random.c`. Here we describe the python interface of the random generator.

This interface is given by a set of functions that can be imported from python module `mmgroup.generators`.

A python function dealing with the randon generator requires a seed. Here a seed is either an object returned by function `rand_make_seed` or `None` (default).

The default seed is (hopefully) thread save, and it is initialized from volatile sources such as the time, the process and the thread id, etc.

A seed created by function `rand_make_seed` is initalized from a fixed source (which is a 64-bit integer).

Each seed may be used by one thread only. In python a seed is crreated for each thread whenever needed.

Python functions in module `mmgroup.generators`

The following functions should be imported from module `mmgroup.generators`.

`.rand_get_seed`()

Return the seed associated with the current thread

This function is used for obtaining a suitable python object for calling a C function that deals with the random generator. The user must not modify that seed!

`.rand_make_seed`(*valu*)

Create a deterministic seed object for the random generator

The function creates a seed object and returns that object. It is intialized with parameter `value`, which must be an unsigned 64-bit integer.

`.rand_bytes_modp`(*p*, *num_bytes*, *seed=None*)

Return a random array of integers mod p

The function returns a one-dimensional numpy array of uniform distributed random integers `x` with `0 <= x < p`. That array has length `num_bytes` and has `dtype = numpy.uint8`.

Parameter `seed` must be either `None` (default, referring to the default seed) or a seed object created by function `rand_make_seed`.

`.rand_fill_bytes_modp`(*p*, *array_bytes*, *seed=None*)

Fill an existing array with random integers mod p

The function fills the one-dimensional numpy array `array_bytes` with uniform distributed random integers `x` with `0 <= x < p`. That array must have `dtype = numpy.uint8`.

Parameter `seed` must be either `None` (default, referring to the default seed) or a seed object created by function `rand_make_seed`.

`.rand_gen_modp`(*p*, *seed=None*)

Return random integer `x` with `0 <= x < p`.

Here 1 <= p <= 2**32 must hold.

Parameter `seed` must be either `None` (default, referring to the default seed) or a seed object created by function `rand_make_seed`.

C interface for file gen_random.c

File `gen_random.c` provides a fast random generator for creating random vectors in the representations of the monster group. This generator supports very fast generation of long random vectors of integers modulo `p` for `p <= 256`.

The seed for such a random generator is given by an array of type `uint64_t seed[MM_GEN_RNG_SIZE]`. In this version we have `MM_GEN_RNG_SIZE = 4`.

The function *`gen_rng_seed(uint64_t *seed)`* fills an existing array `seed` of that type with random seed data. See description of that function for details. The idea is that in a mulithreading environment every thread has its own random generator. There is also a function `gen_rng_seed_no` for a determistic intialization of the `seed`. The user should call function `gen_rng_seed_init` for initialization.

The function *`gen_rng_bytes_modp(uint32_t p, uint8_t *out, uint32_t len, uint64_t *seed)`* writes `len` uniform random numbers `x_i` with `0 <= x_i < p` to the array referred by the pointer `out`. Here `1 < p <= 256` must hold.

This function is optimized for generating large random vectors with, say, `len >= 1000`, as required for the representation of the monster.

Function `gen_rng_modp` generates a single random number `x` with `0 <= x < p` for a 32-bit number `p`.

Internal operation.

We use the xoshiro256** pseudo random number generator, see

https://bashtage.github.io/randomgen/bit_generators/xoshiro256.html

We seed a master generator from various external random sources, e.g. the system time, the process id, etc. We use xoshiro256** for both, the master and the user generator. We shift the master generator by 2**128 steps for seeding a user generator.

Functions

void **`gen_rng_seed_init`**(void)

Seed the master random generator with random data.

Here `seed` is an array of 4 integers of type `uint64_t`.

The function fills the `seed` with random data, using various sources, e.g. the system time, the process id, etc.

void **`gen_rng_seed_no`**(uint64_t *seed, uint64_t seed_no)

Seed a user random generator deterministically.

Here `seed` is an array of 4 integers of type `uint64_t`.

The function fills the `seed` with data depending on the number `seed_no`.

void **`gen_rng_seed`**(uint64_t *seed)

Seed a user random generator from the master random generator.

Here `seed` is an array of 4 integers of type `uint64_t`.

The function seeds the user generator with the seed `seed` from the master random generator. The function is thread-safe if function *`gen_rng_seed_init()`* has been called before.

Each thread must use its own seed.

int32_t **`gen_rng_bytes_modp`**(uint32_t p, uint8_t *out, uint32_t len, uint64_t *seed)

Generate random integers modulo a small number `p`

Generate an array of length `len` of random 8-bit integers `x` with `0 <= x < p`. Here `1 < p <= 256` must hold.

These random integers are written to the array referred by `out`.

Parameter `seed` points to the seed for the random generator. That seed must have been created by function `gen_rng_seed_rnd`` or by a similar function.

The function returns zero in case of success and a nonzero value otherwise.

uint32_t **`gen_rng_modp`**(uint32_t p, uint64_t *seed)

Generate one random integer modulo a number `p`

Generate an integer `x` with `0 <= x < p`. Here `0 <= p < 2**32` must hold. `p = 0` is interpreted as `p = 2**32`.

Parameter `seed` points to the seed for the random generator. That seed must have been created by function `gen_rng_seed_rnd` or by a similar function.

The function returns the random number `x`.

uint64_t **gen_rng_bitfields_modp**(uint64_t p, uint32_t d, uint64_t *seed)

Generate random bit field of integers modulo a number `p`

Generate integers `x_i`, `0 <= i < 64 / d` with `0 <= x < p`. Here parameter `d` is the with of a bit field. The function reuturns an unsigned 64-bit integer `x` with the random value `x_i` stored in bits `i*d + d - 1,..., i *d` of `x`. In case `d = 0` we generate one random number in `x`.

Here `1 <= p < 2**d` must hold. `p = 0` is interpreted as `p = 2**64`.

Parameter `seed` points to the seed for the random generator. That seed must have been created by function `gen_rng_seed_rnd` or by a similar function.

3.4 Description of the `mmgroup.clifford12` extension

3.4.1 Quadratic state vectors

The C functions in modules `qstate.c` and `qsmatrix.c` perform operations on quadratic state matrices given by triples (e, A, Q) as described in **API reference** the section *Long-term stable storage of vectors of the representation*.

A quadratic state vector v of type `qbstate12_type` with component `ncols = n` models a complex vector in a vector space V of dimension 2^n, or an $2^{n-k} \times 2^k$ matrix.

The basis of vector space V is labelled by the elements of the Boolean vector space $\mathbb{F}_2^n$. In C and python programs we represent the element $(x_{n-1}, \ldots, x_0)$ of $\mathbb{F}_2^n$ by the integer $\sum_{0 \leq i < n} 2^i \cdot x_i$. This leads to a natural representation of a vector $v \in V$ as a one-dimensional complex array of length 2^n, starting with index 0.

A quadratic state matrix is a quadratic shape vector augmented by an information about its matrix shape. For more details we refer to the description of struct `qbstate12_type` in file `clifford12.h`.

3.4.2 Header file `clifford12.h`

File `clifford.h` is the header file for shared library `mmgroup_clifford12`. This comprises the following C modules:

$\{0\}$.

Defines

`QSTATE12_UNDEF_ROW`

Marker for an undefined row index in function *qstate12_row_table()*

Enums

enum **`qstate12_error_type`**

Error codes for functions in this module.

Unless otherwise stated, the functions in modules `qstate.c`, `qmatrix.c`, and `xsp2co1.c` return nonnegative in case of success and negative values in case of failure. Negative return values mean error codes as follows:

Values:

enumerator `ERR_QSTATE12_NOTFOUND`

No object with the requested property found.

enumerator `ERR_QSTATE12_INCONSISTENT`

State is inconsistent.

enumerator `ERR_QSTATE12_QUBIT_INDEX`

Qubit index error.

enumerator `ERR_QSTATE12_TOOLARGE`

State is too large for this module.

enumerator `ERR_QSTATE12_BUFFER_OVFL`

Buffer overflow; usually there are now enough rows available.

enumerator `ERR_QSTATE12_Q_NOT_SYMM`

Bit matrix part Q is not symmetric.

enumerator `ERR_QSTATE12_BAD_ROW`

Bad row index for bit matrix.

enumerator `ERR_QSTATE12_INTERN_PAR`

Internal parameter error. Usually a bad row has been requested.

enumerator `ERR_QSTATE12_SCALAR_OVFL`

Overflow or underflow in scalar factor.

enumerator `ERR_QSTATE12_CTRL_NOT`

Bad control_not gate. A qubit in a ctrl-not gate cannot control itself.

enumerator `ERR_QSTATE12_SHAPE_COMP`

Shape mismatch in comparing matrices.

enumerator `ERR_QSTATE12_SCALAR_INT`

Scalar factor is not an integer.

enumerator `ERR_QSTATE12_PARAM`

Parameter error.

enumerator `ERR_QSTATE12_DOMAIN`

Matrix is not is the expected domain.

enumerator `ERR_QSTATE12_SHAPE_OP`

Shape mismatch in matrix operation.

enumerator **ERR_QSTATE12_MATRIX_INV**

Matrix is not invertible.

enumerator **ERR_QSTATE12_PAULI_GROUP**

Matrix is not in the Pauli group.

enumerator **ERR_QSTATE12_NOT_MONOMIAL**

Matrix is not monomial.

enumerator **ERR_QSTATE12_LEECH_OP**

Internal error in operation of group Co_0.

enumerator **ERR_QSTATE12_REP_GX0**

Error in operation of group 2^{1+24}.Co_1.

enumerator **ERR_QSTATE12_NOTIN_XSP**

Element of G_x0 not in 2^{1+24}.

enumerator **ERR_QSTATE12_GX0_TAG**

Bad tag for atom in group G_x0.

enumerator **ERR_QSTATE12_GX0_BAD_ELEM**

Bad element of group G_x0.

struct **qstate12_type**

#include <clifford12.h> Description of a quadratic state matrix.

More precisely, a variable of this type decribes the representation (e', A, Q) of a quadratic mapping $f(e', A, Q)$. Here A is an $(1+m) \times n$ bit matrix. Q is a symmetric $(1+m) \times (1+m)$ bit matrix representing an symmetric bilinear form. We always have $Q_{0,0} = 0$. Then the quadratic mapping $f(e', A, Q) : \mathbb{F}_2^n \rightarrow \mathbb{C}$ is given by

$$f(e', A, Q)(x) = e' \cdot \sum_{\{y=(y_0,\ldots,y_m)\in\mathbb{F}_2^{m+1} \mid y_0=1, y\cdot A=x\}} \exp\left(\frac{\pi\sqrt{-1}}{2} \sum_{j,k=0}^{m} Q_{j,k} y_j y_k\right) .$$

Matrices A and Q are concatenated to an $(m+1) \times (n+m+1)$ matrix M such that $M_{i,j} = A_{i,j}$ for $j < n$ and $M_{i,j} = Q_{i-n,j}$ for $j \geq n$. Matrix M is encoded in a one-dimensional array of unsigned 64-bit integers. Here bit j of entry i of the array corresponds to $M_{i,j}$, with bit 0 the least significant bit.

We do not update column 0 of matrix Q. $Q_{j,0}$ is inferred from $Q_{0,j}$ for $j > 0$ and we always have $Q_{0,0} = 0$.

We also interpret a quadratic mapping f from $\mathbb{F}_2^n$ to $\mathbb{C}$ as a complex $2^{n-k} \times 2^k$ matrix U with entries $U[i,j] = f(b_0, \ldots, b_{n-1})$, where $(b_{n-1}, ..., b_0)_2$, is the binary representation of the integer $i \cdot 2^k + j$, and k is given by component `shape1` of the structure.

The current implementation requires $n + m <= 63$, so that all columns of the bit matrix M fit into a 64-bit integer. With this restriction we may still multiply $2^{12} \times 2^{12}$ matrices U given as quadratic mappings.

A quadratic mapping may be reduced with function *qstate12_reduce()* without notice, whenever appropriate.

Warning:

If an external function changes any component of a structure `qs` of type *`qstate12_type`* (or the content of `qs.data`) then it **must** set `qs.reduced` to zero. The **only** legal way to

set `qs.reduced` to a nonzero value is to call function `qstate12_reduce`. The functions in this module assume that `qs` is actually reduced if `qs.reduced` is not zero.

Public Members

uint32_t **`maxrows`**

No of entries of type `uint64_t` allocated to component `data`

uint32_t **`nrows`**

No $m + 1$ of rows of bit matrix A The value `nrows = 0` encodes the zero mapping.

uint32_t **`ncols`**

No n of columns of bit matrices A and Q.

int32_t **`factor`**

A integer e encoding the complex scalar factor $e' = \exp(e\pi\sqrt{-1}/4) \cdot 2^{\lfloor e/16 \rfloor/2}$

uint32_t **`shape1`**

Describes the shape of the quadratic state matrix as explained above.

uint32_t **`reduced`**

This is set to 1 if the state is reduced and to 0 otherwise.

uint64_t ***`data`**

Pointer to the data bits of the matrix $M = (A, Q)$.

3.4.3 C functions in `bitmatrix64.c`

File `bitmatrix64.c` contains functions for computing with bit matrices. A bit matrix of up to 64 columns is stored in an array of integers of type `uint64_t`. If matrix `m` is stored in the array `a` then entry `m[i,j]` is given by bit `j` of the entry `a[i]` of the array `a`. Here bit `j` has valence 2^j.

In functions dealing with bit matrices we always give the number of rows as a parameter. In some functions we also give the number of columns. In other functions we assume that there are 64 columns, with unused columns ignored.

Functions

uint32_t **`uint64_parity`**(uint64_t v)

Returns the parity of a 64-bit integer `v`.

uint32_t **`uint64_low_bit`**(uint64_t v)

Returns position of lowest bit of a 64-bit integer `v`.

In case `v = 0` the function returns 64.

uint32_t **`uint64_bit_len`**(uint64_t v)

Returns the bit length of a 64-bit integer `v`.

uint32_t **`uint64_bit_weight`**(uint64_t v)

Returns the bit weight of a 64-bit integer `v`.

uint32_t **`uint64_to_bitarray`**(uint64_t v, uint8_t *bl)

Convert 64-bit integer `v` to a bit list.

Return the (sorted) array `bl` of the positions of all bits set in the integer `v`. The function returns the length of the computed array, which is equal to the bit weight of `v`. Array `bl` should have length at least 64. The function is optimized for integers `v` of low bit weight.

void **`bitmatrix64_add_diag`**(uint64_t *m, uint32_t i, uint32_t j)

Add a diagonal matrix to a bit matrix.

Here bit matrix `m` has `i` rows.

We add one to all entries `m[k, j+k]` for `0 <= k < i`.

void **`bitmatrix64_mask_rows`**(uint64_t *m, uint32_t i, uint64_t mask)

Mask rows of a bit matrix.

Here bit matrix `m` has `i` rows.

We replace all rows `m[k]` of `m` by `m[k] & mask`. So we zero the entries `m[i,j]` for all `j` where bit `j` is zero in parameter `mask`.

int32_t **`bitmatrix64_find_masked_row`**(uint64_t *m, uint32_t i, uint64_t mask, uint64_t v)

Find (masked) row of a bit matrix with a certain value.

Here bit matrix `m` has `i` rows.

The function returns the lowest index `k` such that `m[k] & mask == v` holds. The function returns ERR_QSTATE12_NOTFOUND if no such `k` exists.

int32_t **`bitmatrix64_to_numpy`**(uint64_t *m, uint32_t rows, uint32_t cols, uint8_t *a)

Convert bit matrix to numpy array.

Here bit matrix `m` has `rows` rows and `cols` columns.

Then `a` bust be an array of `rows * cols` bytes length. The function writes the bit `m[i,j]` to `a[i * cols + j]`.

The function returns rows * cols in case of success and ERR_QSTATE12_NOTFOUND in case of failure, e.g. if `cols > 64`.

void **`bitmatrix64_from_32bit`**(uint32_t *a32, uint32_t n, uint64_t *a64)

Copy array of 32-bit integers to array of 64-bit integers.

Copy the array `a32` of `n` 32-bit integers to the array `a64` of `n` 64-bit integers. Entries of the array are zero extended.

void `bitmatrix64_to_32bit`(uint32_t *a32, uint32_t n, uint64_t *a64)

Copy array of 64-bit integers to array of 32-bit integers.

Copy the array `a64` of `n` 64-bit integers to the array `a32` of `n` 32-bit integers. Upper 32 bits of the entries are dropped.

uint32_t `bitmatrix64_find_low_bit`(uint64_t *m, uint32_t imin, uint32_t imax)

Find position of lowest bit in a bit matrix.

The function returns the lowest position `k` of a bit in the bit matrix `m`, which is set to one. It scans only the bits at positions `imin <= k < imax`. If all bits at these positions are zero then the function returns `imax`.

Here bit `j` of `m[i]` is considered at position `64*i + j`. Bit matrix `m` should have at least `(imax + 63) >> 6` entries.

void `bitmatrix64_mul`(uint64_t *m1, uint64_t *m2, uint32_t i1, uint32_t i2, uint64_t *m3)

Bit matrix multiplication.

Here `m1` and `m2` have `i1` and `i2` rows, respectively. Only the lowest `i2` columns of `m1` are inspected. Matrix `m2` may have any number of columns, up to 64.

The function computes the matrix product `m1 * m2` in the array `m3`. Matrix `m3` has `i1` rows and the same number of columns as `m2`.

Array `m3` may be equal to array `m1`; but it may not overlap with `m2`.

uint64_t `bitmatrix64_vmul`(uint64_t v, uint64_t *m, uint32_t i)

Multiplication of a vector with a bit matrix.

Here matrix `m` has `i` rows; and `v` is a bit vector. The function returns the product of the bit vector `v` with the bit matrix `m`. If bit vector `v` has `k` leading zero bits then only the lowest `64 - k` rows of matrix `m` are inspected.

int32_t `bitmatrix64_rot_bits`(uint64_t *m, uint32_t i, int32_t rot, uint32_t nrot, uint32_t n0)

Rotate columns of a bit matrix.

Here bit matrix `m` has `i` rows.

For `n0 <= j < n0 + nrot` we map column `j` of matrix `m` to column `n0 + (j + rot) % nrot`. E.g. `nrot = 3, rot = 1, n0 = 0` means that columns are mapped as `0->1, 1->2, 2->0`.

The function returns 0 in case of success and a negative value if any column with index >= 64 is involved in the rotation.

int32_t `bitmatrix64_xch_bits`(uint64_t *m, uint32_t i, uint32_t sh, uint64_t mask)

Exchange columns of a bit matrix.

Here bit matrix `m` has `i` rows.

Exchange column `j` with column `j + sh` of the bit matrix `m`, if bit `j` of `mask` is set. If bit `j` of `mask` is set then bit `j + sh` of `mask` must not be set. No bit of `mask` at position greater or equal to `64 - sh` may be set.

E.g. `qstate12_xch_bits(m, 1, 0x11)` maps columns (…,6,5,4,3,2,1,0) to columns (…,6,4,5,3,2,0,1).

The function returns 0 in case of success and a negative value if any column with index >= 64 is involved in the operation.

int32_t **bitmatrix64_reverse_bits**(uint64_t *m, uint32_t i, uint32_t n, uint32_t n0)

Reverse columns of a bit matrix.

Here bit matrix `m` has `i` rows.

The function reverses `n` columns of the bit matrix `m` starting at column `n0`. So it exchanges bit `m[k, n0 + j]` with bit `m[k, n0 + n - 1 - j]` for `0 <= j < n` and `0 <= k < i`.

The function returns 0 in case of success and a negative value if any column with index >= 64 is involved in the operation.

int32_t **bitmatrix64_t**(uint64_t *m1, uint32_t i, uint32_t j, uint64_t *m2)

Transpose a bit matrix.

Here `m1` is an `i` times `j` bit matrix. Here bit `m1[m,n]` is given by `(m1[m] >> n) & 1`. The function writes the transposed bit matrix of `m1` to `m2`.

uint32_t **bitmatrix64_echelon_h**(uint64_t *m, uint32_t i, uint32_t j0, uint32_t n)

Convert bit matrix `m` to reduced row echelon form.

Matrix `m` has `i` rows. Here we assume that the leading bit of a row of matrix `m` is the most significant bit in that row. For echelonizing `m`, we pivot over nolumns `j0-1,...,j0-n` in that order, ignoring the other columns. We perform row operations (on all possible columns 0,…,63) for changing matrix `m`.

The function returns the number of rows in the (echelonized) matrix `m` that have a nonzero bit in at least one of the pivoted columns.

uint32_t **bitmatrix64_echelon_l**(uint64_t *m, uint32_t i, uint32_t j0, uint32_t n)

Convert bit matrix `m` to reduced row echelon form.

Matrix `m` has `i` rows. Here we assume that the leading bit of a row of matrix `m` is the least significant bit in that row. For echelonizing `m`, we pivot over columns `j0,...,j0+n-1` in that order, ignoring the other columns. We perform row operations (on all possible columns 0,…,63) for changing matrix `m`.

The function returns the number of rows in the (echelonized) matrix `m` that have a nonzero bit in at least one of the pivoted columns.

int32_t **bitmatrix64_cap_h**(uint64_t *m1, uint64_t *m2, uint32_t i1, uint32_t i2, uint32_t j0, uint32_t n)

Compute intersection of row spaces of two bit matrices.

Bit matrix `m1` has `i1` rows, and bit matrix `m2` has `i2` rows. Both matrices, `m1` and `m2`, must be given in reduced echelon form in the sense of function `bitmatrix64_echelon_h`.

The function changes `m1` and `m2` using row operations in such a way that the lower rows of `m1` and `m2` contain the intersection of the rows spaces of the two matrices. For computing that intersection we consider only columns `j0-1,...,j0-n` of matrices `m1` and `m2`. However, we do each row operation on all possible columns `63,...,0`.

The function returns the dimension `n_cap` of the intersection of the row spaces of `m1` and `m2`, ignoring all columns except for columns `j0-1,...,j0-n`.

Assume that the echelonized input matrices `m1` (and `m2`) have `r1` (and `r2`) leading rows that have at least one nonzero bit in columns `j0-1,...,j0-n`, respectively. Then the function changes the leading `r1` rows of `m1` and the leading `r2` rows of `m2` only. These rows are changed in such a way that the last `n_cap` of these rows of the two matrices are equal in columns `j0-1,...,j0-n`. So these last rows contain the intersection of the row spaces of `m1` and `m2``, ignoring all columns but `j0-1,...,j0-n`.

Parameters `j0` and `n` must be the same in this function, and in the previous two calls to function `bitmatrix64_echelon_h` used for echelonizing matrices `m1` and `m2`. This function computes garbage in the arrays `m1` and `m2`, unless both matrices, `m1` and `m2`, have been echelonized in that way before calling this function.

A negative return value `n_cap` means that at least one of the input matrices `m1` and `m2` has not been echelonized correctly. This function may or may not detect an incorrect echelonization of an input matrix.

int32_t **`bitmatrix64_inv`**(uint64_t *m, uint32_t i)

Bit matrix inversion.

Here `m`has `i` rows and `i` columns. `i <= 32` must hold. The function inverts matrix `m` in place.

It returns 0 in case of success and `ERR_QSTATE12_MATRIX_INV` if `m` is not invertible.

int64_t **`bitmatrix64_solve_equation`**(uint64_t *m, uint32_t i, uint32_t j)

Solve a linear bit equation.

Let `M` be the bit matrix with `i` rows and `j` columns stored in the array `m` in the usual way. Let `v` be the column bit vector stored in column `j` of the array `m`.

Then the function tries to solve the equation `transpose(v) * m = x` If such a solution `x` exists then the function returns the transposed vector of the column vector `x` as a nonnegative integer.

The function returns `ERR_QSTATE12_MATRIX_INV` if no solution `x` exists. As a side effect, it echelonizes the lowest `j + 1` columns of matrix `m` with function `bitmatrix64_echelon_h`.

3.4.4 C functions in `uint_sort.c`

File `uint_sort.c` contains functions for sorting arrays of unsigned 32-bit and 64-bit integers.

Functions

void **`bitvector32_copy`**(uint32_t *a_src, uint32_t n, uint32_t *a_dest)

Copy an array to type `uint32_t[]`

Copy the array `a_src` of length `n` to `a_dest`. Any kind of overlap is supported.

void **`bitvector32_heapsort`**(uint32_t *a, uint32_t n)

Sort an array `a` of length `n` with heap sort.

Here `a` is an array of `n` integers of type `uint32_t`. The array is sorted in place. The function is exported for debuging. You should use function `bitvector32_sort`.

void **`bitvector32_sort`**(uint32_t *a, uint32_t n)

Sort an array `a` of type `uint32_t[]` of length `n`

Here `a` is an array of `n` integers of type `uint32_t`. The array is sorted in place.

We use quicksort as the main sorting algorithm. If quicksort reaches a certain recursion level then we switch to heap sort. This guarantees a run time of $O(n\,log(n))$. For small arrays we use insert sort.

void **`bitvector64_copy`**(uint64_t *a_src, uint32_t n, uint64_t *a_dest)

Copy an array to type `uint64_t[]`

Copy the array `a_src` of length `n` to `a_dest`. Any kind of overlap is supported.

void **bitvector64_heapsort**(uint64_t *a, uint32_t n)

Sort an array a of length n with heap sort.

Here a is an array of n integers of type uint64_t. The array is sorted in place. The function is exported for debuging. You should use function bitvector64_sort.

void **bitvector64_sort**(uint64_t *a, uint32_t n)

Sort an array a of type uint64_t[] of length n

Here a is an array of n integers of type uint64_t. The array is sorted in place.

We use quicksort as the main sorting algorithm. If quicksort reaches a certain recursion level then we switch to heap sort. This guarantees a run time of $O(n\,log(n))$. For small arrays we use insert sort.

3.4.5 C functions in qstate12.c

File qstate12.c contains functions for quadratic state matrices as described in the *API reference* in section **Computation in the Clifford group**.

C functions in this module are prefixed with qbstate12_. Unless otherwise stated, these functions return an int32_t, where a nonegative value is interpreted as success, and a negative value is intepreted as failure. Error codes are documented in file clifford12.h.

Typical names for parameters of functions in this module are:

```
Name            | Parameter type
--------------  | ----------------------------------------------
pqs, pqs1, ...  |  Pointer to structure of type qbstate12_type
nqb             |  Number of qubits, i.e. of columns of matrix
                |  A in a structure of type qbstate12_type.
nrows           |  Number of rows of matrix   A  or   Q
                |  in a structure of type qbstate12_type.
i, i1, ...      |  Index of a row of matrix   A  or   Q  ,
                |  starting with 0.
j, j1, ...      |  Index of a column of matrix   A  , with a
                |  column of   A  , corrsesponding to a qubit,
                |  starting with j = 0.
                |  If appropriate, an index  j >= ncols refers
                |  to column (j - ncols) of matrix    Q  .
pv, pv1,...     |  Pointer to a row or column vector of matrix
                |  A, Q  or   M  .
```

Functions

int32_t **qstate12_check**(*qstate12_type* *pqs)

Check a structure of type *qstate12_type*.

Return 0 if ok, or an error code if there is any error in the structure referred by pqs. Part Q[i,0] is set to Q[0, i]. Part Q[0,0] is set ot 0. Irrelevant data bits in valid data rows are zeroed.

int32_t **qstate12_set_mem**(*qstate12_type* *pqs, uint64_t *data, uint32_t size)

Assign memory to a structure of type *qstate12_type*.

Assign the array data of size integers of type uint64_t as memory to structure of type *qstate12_type* referred by pqs.

int32_t **qstate12_zero**(*qstate12_type* *pqs, uint32_t nqb)

Set a structure of type *qstate12_type* to a zero vector.

Set the structure referrred by `pqs` to a zero column vector of length `2**nqb`, corresponding to the zero state of `nqb` qubits.

int32_t **qstate12_vector_state**(*qstate12_type* *pqs, uint32_t nqb, uint64_t v)

Set a structure of type *qstate12_type* to a state vector.

Set the structure referrred by `pqs` to a unit column vector `|v>` of length `2**nqb`. Thus for the quadratic mapping `qs` referred by `pqs` we have `qs(x) = 1` if `x` is equal to the bit vector `v` and `qs(x) = 0` otherwise.

int32_t **qstate12_set**(*qstate12_type* *pqs, uint32_t nqb, uint32_t nrows, uint64_t *data, uint32_t mode)

Set a data in a structure of type *qstate12_type*

Set `pqs->nqb = nqb` so that the state `qs` referred by `pqs` corresponds to a column vector of length `2**nqb`. Set `pqs->nrows = nrows`, and copy `nrows` rows of the array `data` to `pqs->data`.

If `mode == 1` then copy the upper triangular part of the data bit matrix part `Q` to the lower triangular part.

If `mode == 2` then copy the lower triangular part of the data bit matrix part `Q` to the upper triangular part.

Otherwise the part `Q` of the data bit matrix must be symmetric.

The resulting state is checked with function *qstate12_check()*.

int32_t **qstate12_copy**(*qstate12_type* *pqs1, *qstate12_type* *pqs2)

Copy a state of type *qstate12_type* to another state.

Copy the data in the state referred by `pqs1` to the data in the state referred by `pqs2`. Memory must have been allocated to the state `pqs2` with function *qstate12_set_mem()*

int32_t **qstate12_copy_alloc**(*qstate12_type* *pqs1, *qstate12_type* *pqs2, uint64_t *data, uint32_t size)

Copy a state of type *qstate12_type* and allocate mamory.

Equivalent to

```
qstate12_set_mem(pqs2, data, size);
qstate12_copy(pqs1,pqs2);
```

int32_t **qstate12_factor_to_complex**(int32_t factor, double *pcomplex)

Convert a scalar factor to a complex number.

The function takes a scalar `factor` (as given by the component `factor` in the structure `qstate12`) and converts that factor to a complex number. The real part of the result is returned in `pcomplex[0]` and the imaginary part in `pcomplex[1]`.

Caution:

If bit 3 of `factor` is set then the function calculates the complex number 0.

The function returns:

```
 4  if the result is complex, but not real.
```

(continues on next page)

(continued from previous page)

```
3  if the result is real, but not rational.

2  if the result is rational, but not integral.

1  if the result is integral, but not  zero.

0  if the result is zero.

ERR_QSTATE12_SCALAR_OVFL in case of overflow or underflow
```

int32_t **qstate12_factor_to_int32**(int32_t factor, int32_t *pi)

Convert a scalar factor to a 32-bit integer.

The function takes a scalar `factor` (as given by the component `factor` in the structure `qstate12`) and converts that factor to an integer. That integer is stored in `pi[0]`.

Caution:

If bit 3 of `factor` is set then the function calculates the integer zero.

The function returns:

1 if the result is integral, but not zero.

0 if the result is zero.

ERR_QSTATE12_SCALAR_OVFL in case of overflow or underflow

ERR_QSTATE12_SCALAR_INT if the result is not an integer

int32_t **qstate12_conjugate**(*qstate12_type* *pqs)

Conjugate the state referred by `pqs`

The function changes the state referred by `pqs` to its complex complex state.

int32_t **qstate12_mul_scalar**(*qstate12_type* *pqs, int32_t e, uint32_t phi)

Multiply the state referred by `pqs` by a scalar.

The function multiplies the state `qs` referred by `pqs` with the scalar

```
2**(e/2) * exp(phi * pi * sqrt(-1) / 4)  .
```

int32_t **qstate12_abs**(*qstate12_type* *pqs)

Replace entries of a state by their absolute values.

The function replaces the entries of the state referred by `pqs` by their absolute values.

uint64_t **qstate12_get_column**(*qstate12_type* *pqs, uint32_t j)

Return column `j` of the bit matrix `M` stored in `pqs->data` as a bit vector in an integer of type `uin64_t`.

int32_t **qstate12_del_rows**(*qstate12_type* *pqs, uint64_t v)

Delete all rows `i, 1 <= i < pqs->nrows,` from the bit matrix `M` stored in `pqs->data`, if bit `i` in the bit vector `*pv` is set. Here we also adjust the quadratic form `Q` which is part of the bit matrix `M`. Row 0 is never deleted.

int32_t **qstate12_insert_rows**(*qstate12_type* *pqs, uint32_t i, uint32_t nrows)

Insert `nrows` zero rows into the bit matrix `M` stored in `pqs->data`, starting before row `i`. The corresponding zero columns of the quadratic form `Q`, which is part of the bit matrix `M`, are also inserted. `1 <= i <= nrows` must hold. This process multiplies the state vector by the scalar `2**nrows`.

int32_t **qstate12_mul_Av**(*qstate12_type* *pqs, uint64_t v, uint64_t *pw)

Let `M` be the bit matrix stored in `pqs->data`. We return the matrix product `w = A * Transposed(v)`, where `A` is the `A` part of the bit matrix `M`, as a bit vector.

int32_t **qstate12_rot_bits**(*qstate12_type* *pqs, int32_t rot, uint32_t nrot, uint32_t n0)

Rotate qubit arguments of the state `qs` referred by `pqs`.

For `n0 <= i < n0 + nrot` we map qubit `i` to qubit `n0 + (i + rot) % nrot`. E.g. `nrot = 3, rot = 1, n0 = 0` means that bits are mapped as `0->1, 1->2, 2->0`. Let `nn1 = pqs->ncols`. Then the function changes the quadratic mapping `qs` to referred by `pqs` to `qs1` with

```
qs1(x[nn1-1],...,x[n0+nrot],y[nrot-1],...,y[0],x[n0-1],...,x[0])
= qs(x[nn1-1],...,x[n0+nrot],z[nrot-1],...,z[0],x[n0-1],...,x[0]),
```

where `z[j] = y[j - rot (mod 3)]`.

int32_t **qstate12_xch_bits**(*qstate12_type* *pqs, uint32_t sh, uint64_t mask)

Exchange qubit arguments of the state `qs` referred by `pqs`.

Exchange qubit `j` with argument bit `j + sh` of the state `qs` referred by `pqs`, if bit `j` of `mask` is set. If bit `j` of `mask` is set then bit `j + sh` of `mask` must not be set. No bit of `mask` at position greater or equal to `pqs->ncols - sh` may be set.

E.g. `qstate12_xch_bits(pqs, 1, 0x11)` changes the quadratic mapping `qs` to `qs1` with

```
qs1(...,x6,x5,x4,x3,x2,x1,x0) = qs(...,x6,x4,x5,x3,x2,x0,x1) .
```

void **qstate12_pivot**(*qstate12_type* *pqs, uint32_t i, uint64_t v)

Auxiliary function for function *qstate12_reduce()*.

Let `M` be the bit matrix stored in `pqs->data`. The pivoting process is controlled by the bit vector `v`. If `k < i` and bit `k` of `v`is set then row `i` of bit matrix `M`is xored to row `k` of `M`. The columns of the part `Q` of `M` are also adjusted. `1 <= i < pqs->nrows` must hold. Pivoting does not change the state.

For internal use only. Input conditions are not checked.

int32_t **qstate12_sum_up_kernel**(*qstate12_type* *pqs)

Auxiliary function for function *qstate12_reduce()*.

Sum up the kernel of the transformation matrix `A`, which is part of the bit matrix `M = pqs->data`. We assume that `A` is echelonized in the sense that all nonzero rows of `A` are linear independent and that they occur before the zero rows.

int32_t **qstate12_echelonize**(*qstate12_type* *pqs)

Convert the state represented by `pqs` to echelon form.

The representation state referred by `pqs` is converted to (not necessarily reduced) echelon form, and the kernel of bit matrix `A`, which is part of the bit matrix `M = pqs->data` is summed up as described in the guide, section 'Reducing the representation of a quadratic mapping'. The representation of the state is not changed if this is already the case.

This function does not change the state.

int32_t **qstate12_check_reduced**(*qstate12_type* *pqs)

Check if the state represented by `pqs` is reduced.

The function puts `pqs->reduced = 1` if the state is reduced; and it performs no action on `pqs` otherwise. It returns the updated value of `pqs->reduced` (which is 0 or 1) in case of success, and a negative value in case of failure.

This check is much faster than actually performing a reduction. The function may perform some easy reduction steps on the state.

int32_t **qstate12_reduce**(*qstate12_type* *pqs)

Reduce the state represented by `pqs`

The representation state referred by `pqs` is converted to reduced echelon form, and the kernel of bit matrix `A`, which is part of the bit matrix `M = pqs->data` is summed up as described in the guide, section 'Reducing the representation of a quadratic mapping'. The representation of the state is not changed if this is already the case.

This function does not change the state.

int32_t **qstate12_row_table**(*qstate12_type* *pqs, uint8_t *row_table)

Compute a certain table for the state referred by `pqs`

Compute a row table for the state `qs` referred by `pqs`. Here `qs` must be in (not necessarily reduced) echelon form.

The function computes `row_table[j] = i` if the leading bit of row `i` of part `A` of the bit matrix `M = pqs->data` is in column `j`, for `0 <= j < pqs->ncols` and `1 <= i < pqs->nrows`. If now such row exists for column `j` then we put `row_table[j] = QSTATE12_UNDEF_ROW`.

The representation of the state is not changed.

int32_t **qstate12_equal**(*qstate12_type* *pqs1, *qstate12_type* *pqs2)

Check equality of two states.

Return 1 if the states referred by `pqs1` and `pqs2` are equal, 0 if not, and a negative number in case of error. Both states are reduced before comparing them.

int32_t **qstate12_extend_zero**(*qstate12_type* *pqs, uint32_t j, uint32_t nqb)

Insert qubits into a state and set them to 0.

We insert `nqb` zero qubits into the state `qs` referred by `pqs` starting at position `j`.

Let `n = pqs->ncols` so that the state `qs` referred by `pqs` depends on `n` qubits. We change `qs` to the following state `qs1` depending on `n + nqb` qubits:

`qs1(x[n-1],...,x[j],y[nqb-1],...,y[0],x[j-1]...,x[0])` is equal to `qs(x[n-1],...,x[j],x[j-1]...,x[0])` if `y[0] = ... = y[nqb-1] = 0` and equal to zero otherwise. So we increment `pqs->ncols` by `nqb`.

If the input is reduced then the result is also reduced. `pqs->shape1` is set to 0, i.e. a column vector is returned.

int32_t **qstate12_extend**(*qstate12_type* *pqs, uint32_t j, uint32_t nqb)

Insert qubits into a state.

We insert `nqb` qubits into the state `qs` referred by `pqs` starting at position `j`.

Let `n = pqs->ncols` so that the state `qs` referred by `pqs` depends on `n` qubits. We change `qs` to the following state `qs1` depending on `n + nqb` qubits:

`qs1(x[n-1],...,x[j],y[nqb-1],...,y[0],x[j-1]...,x[0])` is equal to `qs(x[n-1],...,x[j],x[j-1]...,x[0])`. So we increment `pqs->ncols` by `nqb`.

The result is not reduced. `pqs->shape1` is set to 0, i.e. a column vector is returned.

int32_t **qstate12_sum_cols**(*qstate12_type* *pqs, uint32_t j, uint32_t nqb)

Sum up the functional values for some qubits.

We sum up `nqb` qubits of the state `qs` referred by `pqs` starting at position `j`.

Let `n = pqs->ncols` so that the state `qs` referred by `pqs` depends on `n` qubits. We change `qs` to the following state `qs1` depending on `n - ncols` qubits:

`qs1(x[n-1],...,x[j+nqb],x[j-1],...,x[0]) = sum_{x[j+nqb-1],...,x[j]} qs1(x[nn1-1],...,x[0])` . So we decrement `pqs->ncols` by `nqb`.

The output is not reduced. `pqs->shape1` is set to 0, i.e. a column vector is returned.

int32_t **`qstate12_restrict_zero`**(*qstate12_type* *pqs, uint32_t j, uint32_t nqb)

Restrict `nqb` qubits starting at postion `j` to 0.

Let `n = pqs->ncols` so that the state `qs` referred by `pqs` depends on `n` qubits. We change `qs` to the following state `qs1` depending on `n` qubits:

`qs1(x[n-1],...,x[0])` is equal to `qs(x[n-1],...,x[0])` if `x[j] = ... = x[j+nqb-1] = 0` and equal to zero otherwise. We do not change the shape of `qs`.

The output is reduced if the input is reduced.

int32_t **`qstate12_restrict`**(*qstate12_type* *pqs, uint32_t j, uint32_t nqb)

Restrict some qubits to 0 and delete them.

Similar to function `qstate12_restrict_zero`, but with deleting the restricted qubits.

Let `n = pqs->ncols` so that the state `qs` referred by `pqs` depends on `n` qubits. We change `qs` to the following state `qs1` depending on `n1 = n - nqb` qubits:

`qs1(x[n1-1],...,x[0])` is equal to `qs(x[n1-1],...,x[j],0,...,0,x[j-1],...,x[0])`. So we decrement `pqs->ncols` by `nqb`.

The output is not reduced. `pqs->shape1` is set to `0`, i.e. a column vector is returned.

In quantum computing theory this operation can be interpreted as measurement of the corresponding qubits with postselection, setting all measured qubits to 0.

int32_t **`qstate12_gate_not`**(*qstate12_type* *pqs, uint64_t v)

Apply a not gate to a state.

Change the state `qs` referred by `pqs` to a state `qs1` with `qs1(x) = qs(x (+) v)`, where `'(+)'` is the bitwise xor operation. The result is not reduced.

Computing `qstate12_gate_not(pqs, 1 << j)` corresponds to negating qubit `j`.

int32_t **`qstate12_gate_ctrl_not`**(*qstate12_type* *pqs, uint64_t vc, uint64_t v)

Apply a contol-not gate to a state.

Change the state `qs` referred by `pqs` to a state `qs1` with `qs1(x) = qs(x (+) <vc,x> * v)`, where `'(+)'` is the bitwise xor operation, and `<.,.>` is the scalar product of bit vectors. The result is not reduced. The scalar product of the bit vectors `j` and `jc` must be zero. Otherwise the `ctrl not` operation is not unitary.

Computing `qstate12_gate_ctrl_not(pqs, 1 << jc, 1 << j)`, for `jc != j`, corresponds to applying a controlled not gate to qubit `j`, contolled by qubit `jc`.

int32_t **`qstate12_gate_phi`**(*qstate12_type* *pqs, uint64_t v, uint32_t phi)

Apply a phase gate to a state.

Change the state `qs` referred by `pqs` to a state `qs1` with `qs1(x) = qs(x) * sqrt(-1)**(phi * <v,x>)`, where `<.,.>` is the scalar product of bit vectors and `'**'` denotes exponentiation. The result is reduced if the input is reduced. Computing `qstate12_gate_ph(pqs, 1 << j, phi)` corresponds to applying a phase `(phi * pi/2)` gate to qubit `j`.

int32_t **qstate12_gate_ctrl_phi**(*qstate12_type* *pqs, uint64_t v1, uint64_t v2)

Apply a controlled phase gate to a state.

Change the state `qs` referred by `pqs` to a state `qs1` with `qs1(x) = qs(x) * (-1)**(<v1,x>*<v2, x>)`, where `<.,.>` is the scalar product of bit vectors and `'**'` denotes exponentiation.

The result is reduced if the input is reduced. Computing `qstate12_gate_ctrl_phi(pqs, 1 << j1, 1 << j2)` corresponds to applying a phase `pi` gate to qubit `j2` controlled by qubit `j1`.

int32_t **qstate12_gate_h**(*qstate12_type* *pqs, uint64_t v)

Apply Hadamard gates to a state.

Apply a Hadamard gate to all qubits `j` of the state `qs` (referred by pqs) with `v & (1 << j) == 1`. Applying a Hadamard gate to gate `j` changes a state `qs` to a state `1/sqrt(2) * qs1`, where `qs1(..,x[j+1],x_j,x[j-1],..)` = `qs(..,x[j+1],0,x[j-1],..)`

- `(-1)**(x_j) * qs(..,x[j+1],1,x[j-1],..)` . The result is not reduced.

3.4.6 C functions in `qstate12io.c`

File `qstate12io.c` contains functions for converting quadratic state matrices as described in the *API reference* in section **Computation in the Clifford group** to complex numbers.

The function in this module are coded according the conventions in module `qstate12.c`.

Functions

int32_t **qstate12_complex**(*qstate12_type* *pqs, double *pc)

Expand a state to an array of complex numbers.

Expand the state `qs` referred by `pqs` to the array referred by the pointer `pc`. The real part of `qs[i]` is stored in `pc[2*i]` and the imaginary part of `qs[i]` is stored in `pc[2*i+1]`. The function reduces `qs`. Here the integer `i` is interpreted as a bit vector as usual.

`pqs->shape1` is ignored. The user has to care for the shape of the returned array. The state `qs` is reduced.

Caution: The function sets `2 * 2**pqs->ncols` entries in the array `pc`.

Return value is as in function `qstate12_entries`.

int32_t **qstate12_entries**(*qstate12_type* *pqs, uint32_t n, uint32_t *v, double *pc)

Convert entries of a state to complex numbers.

The function computes the entries `qs[v[i]]` of the state `qs` referred by `pqs` for `0 <= i < n` and stores these entries in the array `pc`. The real part of `qs[v[i]]` is stored in `pc[2*i]` and the imaginary part is stored in `pc[2*i+1]`. The state `qs` is reduced.

Caution: The function sets `2 * n` entries in the array `pc`.

Depending on the computed matrix entries, the function returns

```
4  if all entries are complex, but not all are real.

3  if all entries are real, but not all are rational.

2  if all entries are rational, but not all are integers.
```

(continues on next page)

(continued from previous page)

```
1  if all entries are integers, but not all are zero.

0  if all entries are zero.
```

A negative return value indicates an error.

int32_t **qstate12_int32**(*qstate12_type* *pqs, int32_t *pi)

Expand a state to an array of 32-bit integers.

Expand the state `qs` referred by `pqs` to the array of integers (of type `int32_t`) referred by the pointer `pi`. Here `qs[i]` is stored in `pi[i]`. The function reduces `qs`. The integer `i` is interpreted as a bit vector as usual.

`pqs->shape1` is ignored. The user has to care for the shape of the returned array. The state `qs` is reduced.

Caution: The function sets `2**pqs->ncols` entries in the array `pc`.

The function returns 0 in case of success, and a negative value in case of error, e.g. if `qs` is not integral or in case of integer overflow.

int32_t **qstate12_to_signs**(*qstate12_type* *pqs, uint64_t *bmap)

Store the signs of a real quadratic state in an array.

Let `qs` be the quadratic state matrix referred by `pqs`. Assume that `qs` is real and has shape `(I, J)`. Let `t[i * 2**J + j] = 0, 1, or 3`, if entry `qs[i, j]` is zero, positive, or negative, respectively. Then we store `t[k]` in bits `2 * l + 1, 2 * l` of `bmap[k >> 5]`, with `l = k & 0x1f`.

The function returns 0 in case of success, and a negative value in case of error, e.g. if `qs` is not a real matrix.

int32_t **qstate12_compare_signs**(*qstate12_type* *pqs, uint64_t *bmap)

Compare signs of a real quadratic state with an array.

Let `qs` and `bmap` as in function `qstate12_to_signs`. The function returns 1 if the signs of `qs` are stored in in `bmap` as described in function `qstate12_to_signs`. Otherwise the function returns 0. The function also returns 0 if matrix `qs` is not real. It returns a negative value in case of an error.

int32_t **qstate12_from_signs**(uint64_t *bmap, int32_t n, *qstate12_type* *pqs)

Construct a state vector from a sign matrix.

Let `bmap` be an array with `2**n` entries representing 0 or (positive or negative) signs as in function `qstate12_to_signs`. Then `bmap` may correspond to a quadratic state vector with entries 0, 1, and -1.

If these enrties correspond to a quadratic state vector `V` (with entries 0, 1, and -1) then the function stores `V` in the quadratic state vector `qs` referred by `pqs` and returns 0. The returned state `qs` is a (column) vector of shape `(0, n)`. If the array `bmap` does not correspond to any quadtatic state vector then the function sets `qs` to the zero vector of shape `(0, n)`and returns -1.

The function returns negative value less than -1 in case of an error.

int32_t **qstate12_mul_matrix_mod3**(*qstate12_type* *pqs, uint64_t *v, uint64_t w)

Compute a certain product modulo 3 with a state matrix.

Let `qs` be the quadratic state matrix referred by `pqs`. Assume that `qs` is rational and has shape `(I, J)`. Let `q` be the row vector of length `2**(I+J)` with `q[i * 2**J + j] = qs[i, j]`.

We consider `v` as a `2**(I+J)` times `32` matrix `M` of integers modulo 3 with `M[k,l]` stored in bits `2*l+1` and `2*l` of entry `v[k]`. We consider `w` as a column vector of `32` integers mod 3 with `w[l]` stored in bits `2*l+1` and `2*l` of the variable `w`.

Then the function returns the matrix product `q * M * v` (modulo 3) as a nonnegative integer less than 3. It returns a negative value in case of error, e.g. if `qs` is not rational.

3.4.7 C functions in `qmatrix12.c`

File `qmatrix12.c` contains basic functions for quadratic

state matrices as described in the **API reference** in section **Computation in the Clifford group**. The functions is this module are based on the functions in module `qstate.c` and on the data structures defined in module `clifford12.h`.

C functions in this module are prefixed with `qbstate12_`. Unless otherwise stated, these functions return an `int32_t`, where a nonegative value is interpreted as success, and a negative value is intepreted as failure. Error codes returned by functions in this module are documented in file `clifford12.h`.

A structure `qs` of type *`qstate12_type`* defines a function $f : \mathbb{F}_2^n \rightarrow \mathbb{C}$ and also a $2^{n-k} \times 2^k$ matrix M with entries $M[i,j] = f(2^k \cdot i + j)$. Here we idenifty the nonegative integers $< 2^n$ with the bit vectors given by their binary representation. For the shape parameters n, k of a state `qs` we have n = `qs.ncols`, k = `qs.qstate1`. Thus indices of vectors and matrices start with 0, as usual in C and python.

While the functions in module `qstate.c` mostly ingnore the shape parameter k, the functions in this module use that shape parameter for determining the shape of a matrix.

Functions

int32_t **`qstate12_std_matrix`**(*qstate12_type* *pqs, uint32_t rows, uint32_t cols, uint32_t rk)

Create a standard matrix with entries 1 in the diagonal.

Create a standard matrix `qs` with `2**rows` rows and `2**cols` columns as an object of type *`qstate12_type`*, and store the result in `*pqs`. Diagonal entries `qs[i,i]` are equal to one for `i < 2**rk`. All other entries of the matrix are zero. `0 <= rk < min(nows, cols)` must hold.

int32_t **`qstate12_unit_matrix`**(*qstate12_type* *pqs, uint32_t nqb)

Create a unit matrix.

Create a unit matrix with `2**nqb` rows and `2**nqb` columns as an object of type *`qstate12_type`*, and store the result in `*pqs`.

int32_t **`qstate12_monomial_column_matrix`**(*qstate12_type* *pqs, uint32_t nqb, uint64_t *pa)

Create matrix with one nonzero entry in each column.

Create a matrix `T` with `2**nqb` rows and `2**nqb` columns as an object of type *`qstate12_type`*, and store the result in `*pqs`.

Matrix `T` is a real `2**nqb` times `2**nqb` transformation matrix which is monomial in the sense that each column contains exactly one nonzero entry 1 or -1. So left multiplication with `T` maps unit vectors to (possibly negated) unit vectors. It transforms a complex column vector `w` of length `2**nqb` to a vector `T * w`.

`pa` refers to an array `a` of integers `a[i]` of length `nqb + 1`. Each integer `a[i]` is interpreted as a bit field via its binary representation. So `a[i,j]` means `(a[i] >> j) & 1`. `a[i, j1:j2]` means the bit field `a[i,j1],...,a[i,j2-1]`.

For any bit vector `v` of length `nqb` let `|v>` be the unit vector with index `v`. For any bit vector `v` of length `nqb + 1` let `|v>` be the (possibly negated) unit vector `(-1)**v[nqb] * |v[0:nqb]>`. `|v1 ^ v2>` and `|1 << v1>` are defined via the corrresponding operators `<<` and `^` in C.

Then `T` maps

```
|0>      to  |a[0, 0:nqb+1]>

|1 << i> to  |a[0, 0:nqb+1] ^ a[i+1, 0:nqb+1]>
```

`T` maps unit vectors to (possibly negated) unit vectors, so `T(v)` is well defined by `|T(v)> = T(|v>)` for a bit field `v` of length `nqb + 1`. We have

```
|T(v1 ^ v2)> = (-1)**Q(v1,v2) * |T(v1) ^ T(v2) ^ T(0)>,
```

for bit fields `v1, v2` of length `nqb + 1` and an alternating bilinear form `Q` depending on the lowest `nqb` bits of `v1` and `v2` only. Thus `T` is defined by the above equation and `Q`. The bilinear form `Q` is defined by:

```
Q(v1, v2) = Q(v2, v1),  Q(v1, v1) = 0,  and

Q(1 << i, 1 << j) =  a[i + 1, j + nqb + 1]``,
for ``0 <= j < i < nqb``.
```

int32_t **qstate12_monomial_row_matrix**(*qstate12_type* *pqs, uint32_t nqb, uint64_t *pa)

Create matrix with one nonzero entry in each row.

Similar to *qstate12_monomial_column_matrix()*; but we create a matrix `T` which is monomial in the sense that each row contains exactly one nonzero entry `1` or `-1`. `qstate12_monomial_row_matrix(*pqs, nqb, *pa)` creates the transposed matrix of `qstate12_monomial_column_matrix(*pqs, nqb, *pa)`.

int32_t **qstate12_monomial_matrix_row_op**(*qstate12_type* *pqs, uint32_t *pa)

Obtain operation of a monomial quadratic state matrix.

A monomial matrix maps unit vectors to unit vectors, if we ignore scalar factors. Let `qs` be the matrix referred by `pqs`. We assume that `qs` has exactly one nonzero entry in each row. Then right multiplication with `qs` maps unit vectors to unit vectors, up to a scalar factor. Otherwrise the function returns ERR_QSTATE12_NOT_MONOMIAL.

If we label each unit vector by a bit vector then the operation of `qs` on these bit vectors labels is affine. If `qs` has shape `(r,c)` then we compute an `r+1` times `c` bit matrix `a` in the array referred by `pa` with the following property:

Label `(b[0],...,b[r-1])` is mapped to label `(1, b[0],...,b[r-1]) * a`.

The function returns the number `r+1` of rows of `a`.

From bit matrix `a` we can construct a unique matrix `qs1`, of the same shape as `qs`, that maps unit vectors to unit vectors as given by the mapping of the labels. In case `r == c` this can be done by calling `qstate12_monomial_row_matrix(pqs1, r, pa)`, where `pqs1` points to a buffer for `qs1`.

Then `qs` can be obtained by multiplying `qs1` with a diagonal matrix.

int32_t **qstate12_mat_reshape**(*qstate12_type* *pqs, int32_t rows, int32_t cols)

Change the shape of a matrix.

Reshape the matrix `T` referred by `pqs` will be a `2**rows` times `2**cols` matrix.

The old shape of the matrix is `(n-k, k)` with `n, k` given by `n = pqs->ncols`, `k = pqs->shape1`. Reshaping must not change the number of entries, so `rows + cols == n` must hold. We follow the convention in the python numpy package for reshaping matrices. Thus index `[i,j]` of a matrix with shape `(I,J)` corresponds to the index `i * 2**J + j` in the one-dimensional array storing that matrix.

If `rows` or `cols` is -1 then it is calculated from the old shape of the matrix. If both, `rows` and `cols`, are -1 then `rows` is set to `0` and `cols` is calculated.

int32_t **qstate12_mat_t**(*qstate12_type* *pqs)

Transpose a matrix in place.

The quadratic state matrix `qs` referred by `pqs` is transposed in place. The result is not reduced.

int32_t **qstate12_mat_trace**(*qstate12_type* *pqs, double *p_trace)

Compute the trace of a quadratic state matrix.

The function computes the trace of the quadratic state matrix `qs` referred by `pqs`. The real part of the trace is stored in `p_trace[0]` and the imaginary part is stored in `p_trace[1]`. The state `qs` is reduced.

Return value is as in function `qstate12_factor_to_complex`.

int32_t **qstate12_mat_itrace**(*qstate12_type* *pqs, int32_t *p_itrace)

Compute the trace of a quadratic state matrix.

Similar to function `qstate12_mat_trace`, buth the trace is stored as an integer in `p_itrace[0]`.

Return value is as in function `qstate12_factor_to_int32`.

int32_t **qstate12_prep_mul**(*qstate12_type* *pqs1, *qstate12_type* *pqs2, uint32_t nqb)

Auxiliary low-level function for function *qstate12_product()*

Prepare the states `qs1` and `qs2` referred by `pqs1` and `pqs2` for matrix multiplication. Here the summation in that operation runs over first `nqb` qubits of `qs1` and `qs2`, regardless of the shape of the input matrices. The function returns a number `row_pos`, so that, after preparation, the first `nqb` columns of submatrices `A1` and `A2` of the bit matrices `M1` and `M2` corresponding to `qs1` and `qs2` will be equal in the following sense:

```
A1[i,j] = A2[i,j]       for i <  'row_pos', j < 'nqb' ,

A1[i,j] = A2[i,j] = 0   for i >= 'row_pos', j < 'nqb' .
```

Also, matrices `A1` and `A2` will both have rank `row_pos - 1`, when excluding row 0 of the two matrices. Some rows of `A1` or `A2` may be deleted to achieve this situation. The result of the summation of the matrix products of `qs1` and `qs2` over the first `nqb` columns (which is used by the matrix multiplication procedure) is not changed by this operation. Apart from this assertion, both states are changed, and they may have less rows than before. They may even be changed to zero, if the result of the multiplication is zero.

The algorithm used here is explained in the **API reference** in section **Multiplication of quadratic mappings**. In the notation in that section the algorithm computes states $\mathrm{qs1}', \mathrm{qs2}'$ with $(\mathrm{qs1}' \odot \mathrm{qs2}')_n = (\mathrm{qs1} \odot \mathrm{qs2})_n$,, where n = `nqb`.

`pqs1->shape1` and `pqs2->shape1` are ignored.

int32_t **qstate12_product**(*qstate12_type* *pqs1, *qstate12_type* *pqs2, uint32_t nqb, uint32_t nc)

Compute a certain product of two states.

Compute a certain product `qs3` of the states `qs1` and `qs2` referred by `pqs1` and `pqs2` and store the (reduced) result in `*pqs1`. Overlap between `pqs1` and `pqs2` is possible. `qs2` is not changed.

Let `n1 = pqs1->ncols, n2 = pqs2->ncols`. Put `qs1a = qstate12_extend(qs, n1, n2-nqb)`, `qs2a = qstate12_extend(qs, nqb, n1-nqb)`. Then `qs1a` and `qs2a` are complex functions on `(nn1 + nn2 - nqb)` bits. Let `qs3a` be the complex function which is the product of the functions `qs1a` and `qs2a`. Then we have `qs3 = qstate12_sum_cols(qs3a, 0, nc)`.

E.g. `qstate12_product(pqs1, pqs2, nc, nc)` is the tensor contraction over the first `nc` qubits of `qs1` and `qs2`. In case `pqs1->ncols = pqs2->ncols = n` the function `qstate12_product(pqs1, pqs2, n, 0)` returns the product of `qs1` and `qs2` (considered as functions). Furthermore, and `qstate12_product(pqs1, pqs2, n, n)` returns the scalar product of `qs1` and `qs2` (considered as vectors).

In general, `qstate12_product(pqs1, pqs2, n, 0)` corresponds to the function $(\mathrm{qs1}' \odot \mathrm{qs2}')_n$ defined in section **Products and tensor products of quadratic mappings** of the **API reference**.

`pqs1->shape1` is set to 0. The user should set `pqs1->shape1` to a reasonable value.

int32_t **`qstate12_matmul`**(*qstate12_type* *pqs1, *qstate12_type* *pqs2, *qstate12_type* *pqs3)

Compute the matrix product of two matrices.

Compute the matrix product `qs3` of the matices `qs1` and `qs2` referred by `pqs1` and `pqs2` and store the (reduced) result in `*pqs3`. Overlap between `pqs1, pqs2, pqs3` is possible.

If `qs1` has shape `(r1,c1)` and `qs2` has shape `(r2,c2)` then `c1 == r2` must hold, and `qs3` has shape `(r1,c2)`

Note that a shape `(r,c)` of a matrix `qs` means a `2**r` times `2**c` matrix, where `r+c == qs.ncols`, `c = qs.shape1`.

int32_t **`qstate12_pauli_vector`**(*qstate12_type* *pqs, uint64_t *pv)

Check if a matrix is in the Pauli group and convert it to a vector.

The **Pauli group** of n qubits is the normal subgroup of the Clifford group of n qubits generated by the not gates, the phase π gates, and by the scalar multiples of the unit matrix by a fourth root of unity.

We represent an element of the Pauli group as a product of $2n + 2$ generators of order 2 or 4. The sequence of these exponents is stored in a bit vector (coded as an integer) as follows:

```
Bit 2n+1:  a scalar factor sqrt(-1)

Bit 2n:    a scalar factor -1

Bit n+i:   a not gate applied to qubit i, 0 <= i < n

Bit i:     a phase pi gate applied to qubit i, 0 <= i < n
```

See section **The Pauli group** in the **API reference** for details.

If the matrix `qs` referred by `pqs` is in the Pauli group of n qubits then the function returns n and stores `qs` as an element of the Pauli group in the vector `v` referred by `pv`. Otherwise the function returns a negative error code.

int32_t **`qstate12_pauli_matrix`**(*qstate12_type* *pqs, uint32_t nqb, uint64_t v)

Convert element of the Pauli group to a matrix.

Here parameter `v` encodes an element of the Pauli group of `nqb` qubits as described in function *qstate12_pauli_vector()*. The function converts `v` to a matrix of shape `(nqb, nqb)` in the Clifford group and stores the result in the state referred by `pqs`.

uint64_t **qstate12_pauli_vector_mul**(uint32_t nqb, uint64_t v1, uint64_t v2)

Multiplication of two elements of the Pauli group.

Here parameters `v1` and `v2`encode two elements of the Pauli group of `nqb` qubits as described in function *qstate12_pauli_vector()*.

The function returns the product `v1 * v2` encoded in the same way.

uint64_t **qstate12_pauli_vector_exp**(uint32_t nqb, uint64_t v, uint32_t e)

Exponentiation of an element of the Pauli group.

Here parameter `v1` encodes an element of the Pauli group of `nqb` qubits as described in function *qstate12_pauli_vector()*.

The function returns the power `v1 ** e` encoded in the same way.

The Pauli group has exponent 4, so `qstate12_pauli_vector_exp(nqb, v1, 3)` returns the inverse of `v1`.

int32_t **qstate12_reduce_matrix**(*qstate12_type* *pqs, uint8_t *row_table)

Perform a special reduction on a quadratic state matrix.

This function performs a special reduction on the quadratic state matrix `qs` referred by `pqs`, as described in the API reference**, section **Reducing a quadratic state matrix**.

This kind of reduction is differnt from the reduction in function *qstate12_reduce()*. It is used internally for computing traces and norms of matrices. It is also used in function *qstate12_pauli_conjugate()*.

The function also computes a table with `pqs->nrows + pqs->ncols` entries in the array referred by parameter `row_table` which is used internally for the operations mentioned above.

int32_t **qstate12_mat_lb_rank**(*qstate12_type* *pqs)

compute the rank of a quadratic state matrix

Let `qs` be the quadratic state matrix referred by `pqs`. The function returns the binary logarithm of the rank of matrix `qs`, which is an integer in case of the nonzero matrix. It returns -1 if `qs` is the zero matrix. A return value less than -1 is an error code. Matrix `qs` is reduced.

int32_t **qstate12_mat_inv**(*qstate12_type* *pqs)

Compute the inverse of a quadratic state matrix.

Let `qs` be the quadratic state matrix referred by `pqs`. The function computes the (reduced) inverse of matrix `qs` in place. It returns ERR_QSTATE12_MATRIX_INV if the matrix is not invertible.

int32_t **qstate12_to_symplectic**(*qstate12_type* *pqs, uint64_t *pA)

Convert quadratic state matrix to a symplectic bit matrix.

Here the quadratic state matrix `qs` referred by `pqs` must be of shape `(k, k)` and invertible. The function computes a `2k` times `2k` bit matrix `A`.

If `v`is a bit vector representing an element of the Pauli group then `v * A` represents the element `qs * v * qs^{-1}` of the Pauli group, up to a scalar factor. Elements of the Pauli group of `k` qubits are encoded as in function `qstate12_pauli_vector()`, ignoring the bits at positions `2k` and `2k + 1`. So `pA[i]` will contain the image of the `i`-th basis vector of the Pauli group, with bits `2k` and `2k + 1` set to zero.

The function returns `2*k` in case of success and a negative value in case of an error. In any case it suffices if the array referred by `pA` has 32 entries.

This function can be considered as a simplified version of function `qstate12_pauli_conjugate`. It computes the natural homomorphism from the group of invertible quadratic state matrices shape `(k, k)` to the symplectic group $\mathrm{S}_{2k}(2)$.

int32_t **`qstate12_to_symplectic_row`**(*qstate12_type* *pqs, uint32_t n)

Conjugate unit vector in Pauli group by quadratic state matrix.

Here the quadratic state matrix `qs` referred by `pqs` must be of shape `(k, k)` and invertible.

Let `v` be the bit vector with bit `n` set and the other bits cleared. Here elements of the Pauli group of `k` qubits are encoded as in function `qstate12_pauli_vector()`, ignoring the bits at positions `2k` and `2k + 1`.

The function returns the element `qs * v * qs^{-1}` of the Pauli group, up to a scalar factor as an integer, encoded in the same way as input vector `v`, ignoring the scalar factor.

Thus in case of success the return value is the same as row `n` of the matrix computes by function `qstate12_to_symplectic_row`. The function returns a negative value in case of an error.

int32_t **`qstate12_pauli_conjugate`**(*qstate12_type* *pqs, uint32_t n, uint64_t *pv, uint32_t arg)

Conjugate Pauli group elements with a Clifford group element.

Here the quadratic state matrix `qs` referred by `pqs` must be of shape `(k,k)` and invertible. The array `v` referred by `pv` has `n` entries `v[0],...,v[n-1]`. Each of these entries is interpreted as an element of the Pauli group of `k` qubits, encoded as in function `qstate12_pauli_vector()`.

The function replaces `v[i]` by the Pauli group element `w[i] = qs * v[i] * qs**(-1)` for all `i < n`. `w[i]` is encoded in the same way as `v[i]`.

Parameter `arg` should usually be a nonzero value. In case `arg == 0` the (complex) argument of the outputs `v[i]` is not computed.

This function uses function `qstate12_reduce_matrix()`. The operation of this function is explained in the **API reference**, section **Reducing a quadratic state matrix** .

3.4.8 Computing in the subgroup G_{x0} of the Monster

The functions in file `xsp2co1.c` implement the group operation of the subgroup G_{x0} (of structure $2^{1+24}.\mathrm{Co}_1$) of the monster.

Represenation of G_{x0} on the tensor product $4096_x \otimes \Lambda$

In [Sey20], section 7.4 and 9, the generators $x_d, x_\delta, y_\delta, x_\pi, \xi$ of G_{x0} are also defined as generators of a group $G_{x1} \subset G(4096_x) \times G(24_x)$. Here $G(4096_x)$ is a subgroup of the real Clifford group $\mathcal{C}_{12}$ operating on the 4096-dimensional real vector space 4096_x, and $G(24_x)$ is the automorphism group Co_0 of the Leech lattice Λ. G_{x1} is a preimage of G_{x0} with $|G_{x1} : G_{x0}| = 2$, and G_{x0} operates faithfully on the tensor product $4096_x \otimes_{\mathbb{Z}} \Lambda$. Let $(x_g, L_g) \in G_{x1} \subset G(4096_x) \times G(24x)$. Component x_g determines component L_g of the pair up to sign. So it suffices to store the image $v_g = v_0 \cdot L_g$ of a fixed shortest vector $v_0 \in \Lambda/3\Lambda$ instead of the whole automorphism L_g of Λ. We put $v_0 = (0, 0, 4, -4, 0, ..., 0)$ in the standard basis of the Leech lattice.

We can reconstruct L_g from x_g and v_g as follows. $G(4096_x)$ has an extraspecial subgroup Q_{x1} of structure 2^{1+24}. The quotient of Q_{x1} by its center $Z(Q_{x1})$ is isomorphic to $\Lambda/2\Lambda$. By definition of G_{x1}, the element L_g operates on $\Lambda/2\Lambda$ in the same way as x_g operates on $Q_{x1}/Z(Q_{x1})$ by conjugation. So that operation of L_g can be reconstructed from x_g. A short vector in $\Lambda/2\Lambda$ has precisely two short preimages in Λ which are opposite. Thus the image of one short vector is known exactly, and the images of all short vectors are known up to sign. Since L_g preserves the scalar product, the images of all short vectors in Λ can be computed. Function `xsp2co1_elem_to_leech_op` in file `xsp2co1.c` constructs L_g from x_g and v_g. The functions in file `gen_leech.c` support operations on $\Lambda/2\Lambda$ and $\Lambda/3\Lambda$.

Embedding a group related to G_{x0} into a Clifford group

The representation 4096_x of the subgroup $G(4096_x)$ of $\mathcal{C}_{12}$ extends to the standard representation of the Clifford group $\mathcal{C}_{12}$. So we may use the theory in section *Long-term stable storage of vectors of the representation* for computing in $G(4096_x)$. In that section the basis vectors of the standard representation of $\mathcal{C}_{12}$ are numbered from 0 to $2^{12} - 1$. We will assign compatible numbers to the basis vectors of the representation 4096_x .

Let $\mathcal{P}$ be the Parker loop as defined in section *The Parker loop*. The vector space 4096_x has basis vectors $d_1^+, d_1^-, d \in \mathcal{P}$, with relations $(-d)_1^\pm = -d_1^\pm$, $(\Omega d)_1^+ = d_1^+$, $(\Omega d)_1^- = -d_1^-$. In the same section the elements of $\mathcal{P}$ are numbered from 0 to $2^{13} - 1$. For $0 \leq d < 2^{11}$ we identify the basis vectors d_1^+ and d_1^- of 4096_x with the basis vectors with numbers corresponding to d and to $d + 2^{11}$ of the representation of $\mathcal{C}_{12}$, respectively.

So we may represent an element x_g of $G(4096_x)$ as a **quadratic state matrix** in a structure of type `qstate12_type`, as defined in file `clifford12.h`. Then we may use the functions in file `qmatrix12.c` for computing in $G(4096_x)$.

The G_{x0} representation

We actually represent an element of G_{x0} in **G_x0 representation**. This is as an array `elem` of 26 integers of type `uint64_t`. That array contains a pair (x_g, v_g) as described above with v_g stored in the first entry `elem[0]`, and x_g stored in the remaining 25 entries of array `elem`.

Vector $v_g \in \Lambda/3\Lambda$ is stored in `elem[0]` as a 48-bit integer in **Leech lattice mod 3 encoding** as described in the documentation of module `gen_leech.c`.

We store the inverse x_g^{-1} of x_g in the upper 25 entries of the array `elem`. The rationale for storing the inverse is that this simplifies the operation of $G(4096)_x$ on Q_{x1} by conjugation, using function `qstate12_pauli_conjugate` in file `qmatrix12.c`. Note that $x_g^{-1} = x_g^\top$, since x_g is orthogonal.

In section *Long-term stable storage of vectors of the representation* an element c of $\mathcal{C}_{12}$ is given as a real $2^{12} \times 2^{12}$ matrix stored in a structure of type `qstate12_type`. In case $c \in G(4096_x)$ that matrix has rational entries, where the denominators are powers of two. A structure of type `qstate12_type` representing a $c \in G(4096_x)$ contains a triple (e, A, Q). There e is a signed integral power of two, and (A, Q) is a pair of bit matrices with up to 25 rows and up to 49 colums, where the first 24 columns belong to matrix A, and the remaining columns belong to matrix Q.

Changing the sign of e corresponds to negation of a matrix $x_g^{-1} \in G(4096_x)$ given by (e, A, Q) or to multiplication by x_{-1}. In the group G_{x0}, changing the sign of v_g has the same effect as changing the sign of x_g; so e can always be made positive. The absolute value of e is just there for scaling the operator norm of a matrix c to 1, and can be omitted.

So we can represent one of the values $\pm x_g^{-1}$ as a pair (A, Q) with an implied positive scalar factor e as above, and negate component v_g of the pair (x_g, v_g), if necessary.

Assuming that x_g^{-1} is stored in a structure `qs` of type `qstate_type`, we copy all valid entries `qs.data[i]` to `elem[i+1]`. This amounts to copying the components (A, Q) of the triple (e, A, Q) from `qs` to `elem`. We always reduce the quadratic state matrix in `qs` before copying it, as indicated in section *Long-term stable storage of vectors of the representation*. We fill unused entries and bits in the array `elem` with zeros. Thus the representation of a $g \in G_{x0}$ in memory is unique. `qs` can easily be reconstructed from `elem`, assuming that the scalar factor e in the triple (e, A, Q) is positive, and that trailing zero entries in the array `elem` are unused.

A word w in the generators of the subgroup G_{x0} of the monster group can be converted to **G_x0 representation** by calling function `xsp2co1_set_elem_word` in file `xsp2co1.c`. Here the word w must be given as an array ` of unsigned 32-bit integers as described in section *Header file mmgroup_generators.h*.

The normal subgroup Q_{x0} of G_{x0}

The group G_{x0} also has an extraspecial normal subgroup Q_{x0} of structure 2^{1+24}. There is a natural isomorphism between the normal subgroup Q_{x1} of $G(4096_x)$ and Q_{x0}. Considering $G(4096_x)$ and $G(24_x)$ as matrix groups, this isomorphism is given by:

$$x \in Q_{x1} \longmapsto x \otimes 1 \in G(4096_x) \otimes G(24_x) \cong G_{x0} \, .$$

We identify Q_{x0} with Q_{x1}. Both groups are generated by elements x_d, x_δ, as described above.

We also represent an element of Q_{x0} as a 25-bit integer in **Leech lattice encoding** as described in section *Description of the mmgroup.generators extension*.

Changing the basis of the space 4096_x

We also use the vectors $(d'), d \in \mathcal{P}$ as basis vectors of 4096_x , where:

$$(d)' = \frac{1}{\sqrt{2}} \left(d_1^+ + d_1^- \right) \, , \quad d_1^\pm = \frac{1}{\sqrt{2}} \left((d)' \pm (\Omega d)' \right) \, .$$

The reason for introducing this basis is that the operation of G_{x0} on Q_{x0} (by conjugation) is easy when we use this basis. Then we obviously have $(-d)' = -(d)'$. We can transform the coordinates of a vector v from the basis given by $d^\pm$ to the basis given by $(d)'$ by multiplying the coordinate vector with a matrix H. Multiplication with matrix H is equivalent to applying the Hadamard gate to Qubit 11 (i.e. to the bit with valence 2^{11}) of the coordinates of v . Thus H is an involution.

The numbering of the 4096 basis vectors $(d)'$ corresponds to the numbering of the positive elemments of $\mathcal{P}$. We may put $(d \oplus 2^{12})' = -(d)'$, where $\oplus$ means bitwise addition modulo 2; then that correspondence holds for all values $0 \leq d < 2^{13}$. The exact defnition of the operator $\oplus$ on the Parker loop $\mathcal{P}$ is given in section *Implementing generators of the Monster group*.

In this basis the operation of the extraspecial group Q_{x0} is very simple:

- $x_e x_{\theta(e)}$ maps $(d)'$ to $(d \oplus e)'$ for $e \in \mathcal{P}$. Here θ is the cocycle, see section *The Parker loop*. In the language of quantum computing this corresponds to a sequence of commuting **not** gates.
- x_ϵ maps $(d)'$ to $(-1)^{\langle d, \epsilon \rangle}(d)'$ for $\epsilon \in \mathcal{C}^*$. Here $\mathcal{C}^*$ is the Golay cocode. In the language of quantum computing this corresponds to a sequence of commuting **phase**-π gates.

Since H is in $\mathcal{C}_{12}$, it operates on Q_{x0} by conjugation. The following subtlety has to be considered in function `conv_pauli_vector_xspecial` (in file `xsp2cco1.c`) implementing that operation. H exchanges the anticcommuting elements of Q_{x0} of with numbers `0x800` and `0x800000` in Leech lattice encoding. Thus it negates the element with number `0x800800`.

Converting an element in G_x0 representation to a word of generators

Perhaps the most inportant function in this module is function `xsp2co1_elem_to_word`. This function converts an element of the subgroup G_{x0} from **G_x0 representation** to a word w of generators of the monster group. That word w is unique and **reduced** in the sense explained in the description of that function.

There is also a companion function `xsp2co1_reduce_word` that first converts an arbitrary word in the generators of the subgroup G_{x0} of the monster to **G_x0 representation**. Then it calls function `xsp2co1_elem_to_word` to compute the **reduced** word equal to the input word.

We briefly explain the implementation of function `xsp2co1_elem_to_word`. Let $g \in G_{x0}$ be given in **G_x0 representation**. The function first computes the image $g^{-1} x_\Omega g$ of x_Ω using function `xsp2co1_xspecial_conjugate`, with Ω as in [Sey20]. Using that image and function `gen_leech2_reduce_type4` in file `gen_leech.c` we can compute a word w_1 in the generators of G_{x0} such that $g_1 = g \cdot w_1$ stabilizes x_Ω up to sign.

Then g_1 is in the monomial subgroup N_{x0} of G_{x0} of structure $2^{1+24}.2^{11}.M_{24}$. For any $g_1 \in N_{x0}$ function `xsp2co1_elem_monomial_to_xsp` computes a word in the generators of the monster that is equal to g_1.

Given a monomial element g_1 in **G_x0 representation**, function `xsp2co1_elem_monomial_to_xsp` computes a word of generators equal to g_1 as follows. Ignoring signs and identifying the basis vector d_1^+ of 4096_x with the basis vector d_1^-, Table 3 in [Sey20] states that $x_\pi y_e x_f$ maps $d_1^\pm$ to $(d^\pi ef)_1^\pm$. This corresponds to an affine mapping $d \mapsto d^\pi ef$ from $\mathcal{C}/\langle\Omega\rangle$ to itself, from which we can easily compute e, f (modulo sign and Ω), and also π. Thus we can find a word w_2 in the generators $x_\pi y_e x_f$, such that $g_2 = g_1 w_2$ stabilizes all basis vectors of 4096_x up to sign, and possibly exchanging d_1^+ with d_1^-. Then g_2 can easily be converted into a word in the generators $x_{-1}, x_\Omega, x_\delta, \delta \in \mathcal{C}^*$.

We use function `qstate12_monomial_matrix_row_op` in file `qmatrix12.c` for obtaining the affine mapping $d \mapsto d^\pi ef$ from the **G_x0 representation** of g_1. We use the functions in file `mat24_functions.c` for obtaining π from that affine mapping.

3.4.9 C functions in `xsp2co1.c`

File `xsp2co1.c` contains functions for computing in the subgroup G_{x0} (of structure $2^{1+24}.\mathrm{Co}_1$) of the monster.

Functions

uint64_t **`xsp2co1_find_chain_short_3`**(uint64_t v3_1, uint64_t v3_2)

Find vector not orthogonal to vectors in Leech lattice mod 3.

Given two vectors $v_{3,1}$ and $v_{3,2}$ in the Leech lattice mod 3, the function returns a vector $v_{3,3}$ such that the scalar product $\langle v_{3,3}, v_{3,i}\rangle$ is not zero for both, $i = 1, 2$. $v_{3,3}$ has precisely two nonzero coordinates and is hence short. Such a vector $v_{3,3}$ exists if none of the vectors $v_{3,1}, v_{3,2}$ is zero. If no such vector $v_{3,3}$ exists then the function returns 0.

All vectors $v_{3,i}$ are in given in **Leech lattice mod 3 encoding**.

int32_t **`xsp2co1_chain_short_3`**(*qstate12_type* *pqs, uint32_t n, uint64_t *psrc, uint64_t *pdest)

Apply transformation in G_{x0} to vectors in Leech lattice mod 3.

Let $x_g \in G(4096_x)$ be given by the quadratic state matrix referred by `pqs`. Let `psrc` be an array $(v_0, \ldots v_{n-1})$ of n short vectors in the Leech lattice mod 3, given in **Leech lattice mod 3 encoding**. We try to compute $w_i = v_i x_g$ for $0 \leq i < n$ and to store w_i in `pdest[i]`, also in **Leech lattice mod 3 encoding**. Unfortunately, the short vector w_i is defined to sign only. In other words, x_g may correspond to two different elements $\pm g$ in the automorphisem group Co_0 of the Leech lattice.

To specify the correct $g \in \mathrm{Co}_0$, we must store the correct value $w_0 = v_0 x_0$ in `pdest[0]` and provide short vectors v_i such that the scalar product $\langle v_{i-1}, v_i\rangle$ of adjacent vectors is not zero for $i > 0$. The the correct values $v_i x_i$ are determined by ortohonality of g.

The function returns garbage if the input conditions for the x_g, v_0, w_0 are not satisfied. It returns a negative value if two adjacent vectors v_0 are orthognal or not short.

int32_t **`xsp2co1_elem_to_qs_i`**(uint64_t *elem, *qstate12_type* *pqs)

Get component $x_g^{-1} \in G(4096_x)$ from $g \in G_{x0}$.

Let $g \in G_{x0}$ be stored in the array `elem`, in **G_x0 representation**. This means that g is given as a pair $(x_g, v_g) \in G(4096_x) \times \Lambda/3\Lambda$. The function stores x_g^{-1} in the structure `qs` of type `qstate12_type` referred by `pqs`.

Component v_g is equal to `elem[0]` (in **Leech lattice mod 3 encoding**).

Caution:

This is a low-level function. After returning, the structure `qs` and the array `elem` share the same data block.

Caution:

Internally, we store the inverse of component x_g in the element g, and this function also stores that inverse in the structure `qs`.

int32_t **`xsp2co1_elem_to_qs`**(uint64_t *elem, *qstate12_type* *pqs)

Get component $x_g \in G(4096_x)$ from $g \in G_{x0}$.

Let $g \in G_{x0}$ be stored in the array `elem`, in **G_x0 representation**. This means that g is given as a pair $(x_g, v_g) \in G(4096_x) \times \Lambda/3\Lambda$. The function stores x_g in the structure `qs` of type *`qstate12_type`* referred by `pqs`. x_g represents a 4096×4096 matrix.

Caution:

The structure referred by `pqs` must provide sufficient memory for data, see function `qstate12_set_mem` in file `qstate12.c`; here `pqs->data` should be at least 25.

int32_t **`xsp2co1_qs_to_elem_i`**(*qstate12_type* *pqs, uint64_t v_g, uint64_t *elem)

Construct $g \in G_{x0}$ from pair (x_g, v_g).

The function constructs a $g \in G_{x0}$ as a pair $(x_g, v_g) \in G(4096_x) \times \Lambda/3\Lambda$ and stores the result in the array `elem` in **G_x0 representation**. The value $x_g^{-1} \in G(4096_x)$ must be given as a structure of type *`qstate12_type`* referred by `pqs`. The value $v_g \in \Lambda/3\Lambda$ must be given by parameter `v3` in **Leech lattice mod 3 encoding**.

Caution:

As a low-level function, this function may construct a value g which is not in G_{x0}. Function `xsp2co1_set_elem_word` should be used for constructing an element of G_{x0} instead.

Caution:

Internally, we store the inverse of component x_g in the element g, and this function also requires that inverse in the structure referred by `pqs`.

int32_t **`xsp2co1_reduce_elem`**(uint64_t *elem)

Reduce an $g \in G_{x0}$ to a standard form.

Let $g \in G_{x0}$ be stored in the array `elem`, in **G_x0 representation**. This means that g is stored as a pair $(x_g, v_g) \in G(4096_x) \times \Lambda/3\Lambda$. The function reduces the components x_g, v_g to their standard form in place.

In functions that construct elements of G_{x0} these components are reduced automatically.

void **`xsp2co1_neg_elem`**(uint64_t *elem)

Negate an $g \in G_{x0}$.

Let $g \in G_{x0}$ be stored in the array `elem`, in **G_x0 representation**. The function negates g in place.

Negation is equivalent to multiplication with the generator x_{-1}, and also to multiplication with the generator y_Ω.

void **`xsp2co1_copy_elem`**(uint64_t *elem1, uint64_t *elem2)

Copy a $g \in G_{x0}$.

The function copies the element of G_{x0} stored in the array `elem1` (in **G_x0 representation**) to the array `elem2`.

int32_t **xsp2co1_elem_to_bitmatrix**(uint64_t *elem, uint64_t *pA)

Map element of G_x0 to a bit matrix.

The function maps the element of G_{x0} stored in

the array `elem` (in **G_x0 representation**) to a 24×24 bit matrix A, which will be stored in the array referred by `pA`. Bit matrix A operates on a vector on the Leech lattice modulo 2 (in Leech lattice encoding) by right multiplication.

So this function computes the homomorphism from the group G_{x0} onto the group Co_1.

int32_t **xsp2co1_mul_elem**(uint64_t *elem1, uint64_t *elem2, uint64_t *elem3)

Multiply two elements of the group G_{x0}.

Let $g_1, g_2 \in G_{x0}$ be stored in the arrays `elem1`, `elem2` in **G_x0 representation**. The function computes $g_1 \cdot g_2$ and stores the result in the array `elem3` in **G_x0 representation**.

Any kind of overlapping beween the arrays `elem1`, `elem2`, and `elem3` is allowed.

int32_t **xsp2co1_inv_elem**(uint64_t *elem1, uint64_t *elem2)

Invert an element of the group G_{x0}.

Let $g_1 \in G_{x0}$ be stored in the array `elem1`, in **G_x0 representation**. The function computes g_1^{-1} and stores the result in the array `elem2` in **G_x0 representation**.

Any kind of overlapping beween the arrays `elem1` and `elem2` is allowed.

int32_t **xsp2co1_conj_elem**(uint64_t *elem1, uint64_t *elem2, uint64_t *elem3)

Conjugate elements of the group G_{x0}.

Let $g_1, g_2 \in G_{x0}$ be stored in the arrays `elem1`, `elem2` in **G_x0 representation**. The function computes $g_2^{-1} \cdot g_1 \cdot g_2$ and stores the result in the array `elem3` in **G_x0 representation**.

Any kind of overlapping beween the arrays `elem1`, `elem2`, and `elem3` is allowed.

int32_t **xsp2co1_xspecial_conjugate**(uint64_t *elem, uint32_t n, uint64_t *ax, uint32_t sign)

Conjugation of elements of Q_{x0} with an element of G_{x0}.

Let $x_0, \ldots, x_{n-1}$ a list of n elements of Q_{x0} stored in the the array `ax` in **Leech lattice encoding**. Let $g \in G_{x0}$ be stored in the array `elem` in **G_x0 representation**.

Then the function replaces the element x_i by $g^{-1} x_i g$ for $0 \leq i < n$.

Parameter `sign` should usually be a nonzero value. In case `sign = 0` the signs of the returned vectors are not computed.

int32_t **xsp2co1_xspecial_img_omega**(uint64_t *elem)

Conjugation of Ω with an element of G_{x0}.

Let $g \in G_{x0}$ be stored in the array `elem` in **G_x0 representation**, and let Ω be the standard frame in the Leech latice. The function returns $g^{-1} \Omega g$ in **Leech lattice encoding**, ignoring the sign of the result.

int32_t **xsp2co1_elem_check_fix_short**(uint64_t *elem1, uint32_t v)

Check if an element of G_{x0} fixes a short vector.

This function is deprecated!

Let v be the short vector in the Leech lattice modulo 2 given by parameter `v` in **Leech lattice encoding**. Let $g \in G_{x0}$ be stored in the array `elem` in **G_x0 representation**.

If v is short then it corresponds to a vector v' in the 24-dimensional Leech lattice Λ up to sign. Note that the operation of G_{x0} on Λ is also defined up to sign only. So in the general case we cannot distinguish whether g fixes or negates v'.

However, G_{x0} has a (faithful) representation on the tensor product $\Lambda \otimes 4096_x$ for a certain representation 4096_x, see [Con85] for details. Here the operation of G_{x0} on 4096_x is also defined up to sign only.

If the character of g on 4096_x is nonzero then we can flip the signs in both representations, Λ and 4096_x, without changing the tensor product. Thus requiring the character of g on 4096_x to be positive determines the sign of the operation of g on Λ uniquely. In this case we return 0 if g fixes the vector v', and 1 if g negates v'. Otherwise we return 6.

If the character of g on 4096_x is zero then there is no way to determine the sign of the operation of g on Λ. In this case we return 2 if g fixes the vector v' (up to sign) and 6 otherwise.

In cases where g and $-g$ are in different classes in G_{x0} we may sometimes obtain the class of g by using this function.

We return a negative value in case of any error. It is an error if v' is not short, i.e. v is not of type 2.

int32_t **`xsp2co1_xspecial_vector`**(uint64_t *elem)

Convert $x \in Q_{x0}$ from G_{x0} rep to Leech.

Let $x \in Q_{x0} \subset G_{x0}$ be stored in the array `elem` in **G_x0 representation**. The function returns x as an integer in **Leech lattice encoding**.

The function returns a negative number in case of error. E.g. in case $x \notin Q_{x0}$ it returns `ERR_QSTATE12_NOTIN_XSP`.

void **`xsp2co1_unit_elem`**(uint64_t *elem)

Store neutral element of G_{x0}.

The function stores the neutral element of G_{x0} in the array `elem` in **G_x0 representation**.

uint32_t **`xsp2co1_is_unit_elem`**(uint64_t *elem)

Check if `elem` is neutral element of G_{x0}.

The function returns 1 if `elem` is the neutral element of G_{x0} and 0 otherwise.

int32_t **`xsp2co1_elem_xspecial`**(uint64_t *elem, uint32_t x)

Convert $x \in Q_{x0}$ from Leech to G_{x0} rep

Let $x \in Q_{x0} \subset G_{x0}$ be stored in parameter `x` in **Leech lattice encoding**. The function converts x to **G_x0 representation** and stores the result in the array `elem` .

int32_t **`xsp2co1_mul_elem_word`**(uint64_t *elem, uint32_t *a, uint32_t n)

Right multiply an element of G_{x0} with a word of generators.

Let $g \in G_{x0}$ be stored in the array `elem` in **G_x0 representation**. We replace g by $g \cdot w$, where w is a word in the generators of G_{x0} of length n. w is stored in the array `a`, and each entry of `a` encodes a generator of G_{x0} as described in file `mmgroup_generators.h`.

The function fails and returns ERR_QSTATE12_GX0_TAG if not all atoms of the word w are in G_{x0}.

int32_t **`xsp2co1_set_elem_word`**(uint64_t *elem, uint32_t *a, uint32_t n)

Convert word of generators of G_{x0} to G_x0 representation.

Let w be a word in the generators of G_{x0} of length n. w is stored in the array `a`, and each entry of `a` encodes a generator of G_{x0} as described in file `mmgroup_generators.h`. We convert the word w to an element of G_{x0} in **G_x0 representation** and store the result in the array `elem`.

The function fails and returns ERR_QSTATE12_GX0_TAG if not all atoms of the word w are in G_{x0}.

int32_t **xsp2co1_mul_elem_atom**(uint64_t *elem, uint32_t v)

Right multiply an element of G_{x0} with a generator.

Equivalent to `xsp2co1_mul_elem_word(elem, &v, 1)`

int32_t **xsp2co1_set_elem_atom**(uint64_t *elem, uint32_t v)

Convert a generator of G_{x0} to G_x0 representation.

Equivalent to `xsp2co1_set_elem_word(elem, &v, 1)`

int32_t **xsp2co1_set_elem_word_scan**(uint64_t *elem, uint32_t *a, uint32_t n, uint32_t mul)

Convert word of generators of G_{x0} to G_x0 representation.

Parameters and operation are as in function `xsp2co1_set_elem_word`. But in contrast to function `xsp2co1_set_elem_word`, this function succeeds also if just a prefix of the word `a` is in the subgroup G_{x0}.

Let `k` be the greatest number such that all prefixes of `a` of length at most `k` are in the group G_{x0}. Let a_k be the element of G_{x0} corresponding to the prefix of `a` of length `k`.

If parameter `mul` is zero then we convert the word a_k to an element of G_{x0} in **G_x0 representation** and store the result in the array `elem`. Otherwise we multiply the element `elem` with the word a_k and store the array in `elem`.

3.4.10 C functions in `xsp2co1_word.c`

File `xsp2co1_word.c` contains additional functions for computing in the subgroup G_{x0} (of structure $2^{1+24}.\mathrm{Co}_1$) of the monster. This file can be considered as a supplement to file `xsp2co1.c`.

Functions

uint64_t **xsp2co1_to_vect_mod3**(uint64_t x)

Auxiliary function for function `xsp2co1_add_short_3_leech`

The function converts a vector in $\Lambda/3\Lambda$ from **Leech lattice mod 3** encoding to the encoding to the encoding of a vector in $(\mathbb{Z}/3\mathbb{Z})^{24}$ used in the `mmgroup.mm_op` extension.

uint64_t **xsp2co1_from_vect_mod3**(uint64_t x)

Inverse of function `xsp2co1_to_vect_mod3`

The function converts a vector in $\Lambda/3\Lambda$ from the encoding of a vector in $(\mathbb{Z}/3\mathbb{Z})^{24}$ used in the `mmgroup.mm_op` extension to the **Leech lattice mod 3** encoding.

int32_t **xsp2co1_elem_to_leech_op**(uint64_t *elem, int8_t *pdest)

Get Leech lattice matrix from $g \in G_{x0}$.

Let $g \in G_{x0}$ be stored in the array `elem`, in **G_x0 representation**. G_{x0} operates faithfully on the space $4096_x \otimes_{\mathbb{Z}} \Lambda$. This function constructs a 24×24 integer matrix L_g such that $\frac{1}{8} L_g$ corresponds to the operation of g on Λ. It stores entry $L_g[i,j]$ in `dest[24*i+j]`. Matrix L_g is unique up to sign.

Function `xsp2co1_elem_to_qs(elem,...)` computes a (representation of) an orthogonal 4096×4096 matrix x_g such that right multiplication with the Kronecker product $\frac{1}{8} x_g \otimes L_g$ is equal to the action of g on $4096_x \otimes \Lambda$.

int32_t **xsp2co1_short_3_to_leech**(uint64_t x, int8_t *pdest)

Compute integral short Leech lattice vector from vector mod 3.

Given a short Leech lattice vector `x` (modulo 3) in **Leech lattice mod 3** encoding, the function computes the real coordinates of vector `x` in the array referred by `pdest`. `pdest` must have length 24. As usual, the norm (i.e. the squared sum of the coordinates) of the computed short vector is normalized to 32.

The function returns 0 if `x` encodes a short Leech lattice vector mod 3 and a negative value otherwise.

int32_t **xsp2co1_short_2_to_leech**(uint64_t x, int8_t *pdest)

Compute integral short Leech lattice vector from vector mod 2.

Given a short Leech lattice vector `x` (modulo 2) in **Leech lattice** encoding, the function computes the real coordinates of vector `x` in the array referred by `pdest`. `pdest` must have length 24. As usual, the norm (i.e. the squared sum of the coordinates) of the computed short vector is normalized to 32. Note that the result is defined up to sign only. Here the function chooses an arbitrary sign.

The function returns 0 if `x` encodes a short Leech lattice vector mod 2 and a negative value otherwise.

int32_t **xsp2co1_elem_monomial_to_xsp**(uint64_t *elem, uint32_t *a)

Map monomial element of G_{x0} to element of Q_{x0}.

Let $g \in G_{x0}$ stored in the array `elem`. The matrix corresponding to g in the representation 4096_x must be monomial. The function computes a word w of in the generators of G_{x0} such that $gw \in Q_{x0}$. The word w has length at most 2 and is stored in the array `a`. Each entry of `a` encodes a generator of G_{x0} as described in file `mmgroup_generators.h`. The function returns the length of that word.

The atoms in the word have tags `p, y` in that order. Each word is stored as the inverse of a generator.

int32_t **xsp2co1_elem_to_word**(uint64_t *elem, uint32_t *a)

Convert element of G_{x0} to a word in its generators.

Let $g \in G_{x0}$ be stored in the array `elem`. The function converts g to a **reduced** word in the generators of g and stores that word in the array `a`. Then each entry of `a` encodes a generator of G_{x0} as described in file `mmgroup_generators.h`. The function returns the length of that word.

The **reduced** word stored in the array `a` may have up to 10 entries. The tags of the entries in that word are `xdyplplplp` in that order. See documentation of class `mmgroup.MMGroup` for the meaning of these tags. Each entry of a word may encode the neutral element as a generator; then that entry is dropped. We assert that the number of entries with tag `l` is minimal.

int32_t **xsp2co1_reduce_word**(uint32_t *a, uint32_t n, uint32_t *a1)

Reduce a word of generators of G_{x0}.

Let $g \in G_{x0}$ be stored in the array `a` as a word w of length n. The function computes the **reduced** word w_1 equal to w in the array `a1` and returns the length of the reduced word. Legal tags for the word w are `d`, `x`, `y`, `p`, and `l`. See documentation of class `mmgroup.MMGroup` for the meaning of these tags.

It uses function `xsp2co1_elem_to_word` for computing w_1. The word w_1 stored in the array `a1` may have up to 10 entries. Arrays `a` and `a1` may overlap.

int32_t **xsp2co1_elem_subtype**(uint64_t *elem)

Return the subtype of an element of G_{x0}.

Let $g \in G_{x0}$ be stored in the array `elem`. The function returns the subtype of g. If g maps the standard frame Ω of the Leech lattice modulo 2 to a frame of subtype t then g has subtype t.

The subtype is returned as an integer as in function `gen_leech2_subtype` in module `gen_leech.c`.

Since the subtype is determined by the size of the denominators of the representation 4096_x, it can be computed very fast.

The function returns -1 in case of an error.

uint32_t **xsp2co1_check_word_g_x0**(uint32_t *w, uint32_t n)

Check if a word of generators of the monster is in G_{x0}.

We check if the word `w` of length `n` of generators of the monster group is in the subgroup G_{x0}. The function returns the following status information:

0: `w` is in G_{x0}

1: `w` is not in G_{x0}

2: Nothing is known about `w`

Words of generators of the monster are implemented as described in file `mmgroup_generators.h`.

int32_t **xsp2co1_isotropic_type4**(uint32_t v, uint64_t *pB, int32_t n)

Compute maximal isotropic space corresponding to a type-4 vector.

Any type-4 vector in the Leech lattice mod 2 corrsponds to a unique maximal isotropic space (of dimension 12) in the Leech lattice mod 2. E.g. the standard type-4 vector Ω corresponds to the space spanned by Ω and (the images in the Leech lattice mod 2 of) all even Golay cocode words.

Given a type-4 vector v in *Leech lattice encoding*, we usually want to find the intersection of the isotropic space $v^{(\perp)}$ corresponding to v with a given linear subspace X of the Leech lattice mod 2. Let X be the space spanned by the vectors $b_0, ..., b_{n-1}$ in the array `B` referred by `pB` of size `n`.

The function modifies the basis in the array `B`, so that it will be the a basis of the space $v^{(\perp)} \cup X$ of dimension `m` in reduced echelon form. The function returns `m` in case of success and a negative value in case of failure. The function fails if v is not of type 4.

In case $n < 0$ we assume that `B` is a basis of the whole Leech lattice mod 2 and return a basis of $v^{(\perp)}$ in `B`. Array `B` must have size at least $\max(n, 12)$.

int32_t **xsp2co1_elem_row_mod3**(uint64_t *elem, uint32_t column, uint64_t *v)

A low-level function to be used for testing.

A projection matrix Π is a symmetric matrix with one eigenvalue 1 and the other eigenvalues equal to zero operating on an Euclidean vector space. Let $g \in G_{x0}$ be stored in the array `elem` in **G_x0 representation**. This function left multiplies g by a certain projection matrix Π. The result $y = \Pi \cdot g$ is an element of the vector space $4096_x \otimes 24_x$. The function reduces the coordinates of y modulo 3 and stores the result in the array `v` in a format compatible the format used in the `mmgroup.mm_op` extension.

Right multiplcation of g by G_{x0} commutes with left multiplication of g by Π, so that we can test the right multiplication by G_{x0} implemented in this module against the corresponding multiplication implemented in the `mmgroup.mm_op` extension. This leads to the important interoperability test in the python function `mmgroup.tests.test_clifford.test_xs1_vector.test_vector`.

We specify the projection matrix Π as a tensor product $\Pi_{4096} \otimes \Pi_{24}$. Here Π_{24} projects onto the fixed short Leech lattice vector $(0, 0, 1, -1, 0, \ldots, 0)$. Π_{4096} is the projection onto the coordinate with number `column` of the space 4096_x.

Remark:

The result is an array with 4096 entries corresponding to the entries with tags `Z` and `Y` of a vector in the represention ρ_3, as described in section **The Representation of the Monster Group** of the **API reference**.

Warning:

This function works only if the data type `uint_mmv_t` used in the `mmgroup.mm_op` extension is equal to the data type `uint64_t`.

int32_t **`xsp2co1_elem_read_mod3`**(uint64_t *v, uint64_t *elem, uint32_t row, uint32_t column)

Read entry of a transfromed vector of the monster rep modulo 3.

Let $g \in G_{x0}$ be stored in the array `elem`. Let v be the vector of the representation $4096_x \otimes 24_x$ modulo 3 stored in the array `v` in a format compatible to the format used in the `mmgroup.mm_op` extension. Then the function returns the entry of the vector $v' = v \cdot g^{-1}$ in row `0 <= row < 4096` and column `0 <= column < 24` of v'.

This function is considerably faster than the computation of $v' = v \cdot g^{-1}$ using the functions in the `mmgroup.mm_op` extension.

In case `column = 24` the function returns the value `(v'[row,2] - v'[row,3]) mod 3`. In case of success the return value is a nonnegative integer less than 3. A negative return value indicates failure.

3.4.11 C functions in `xsp2co1_elem.c`

File `xsp2co1_elem.c` contains functions for analyzing elements of the subgroup G_{x0} (of structure $2^{1+24}.\mathrm{Co}_1$) of the monster.

Functions

int32_t **`xsp2co1_elem_to_N0`**(uint64_t *elem, uint32_t *g)

Convert element of G_{x0} to element of N_0.

Let $g \in G_{x0}$ be stored in the array `elem1` in **G_x0 representation**. The function converts g to an element N_0, as described in the documentation of file `mm_group_n.c`. The result is stored in the array of length 5 referred by parameter `g`.

The function returns 0 in case of success and `ERR_QSTATE12_GX0_BAD_ELEM` if g is not and N_0.

int32_t **`xsp2co1_elem_from_N0`**(uint64_t *elem, uint32_t *g)

Convert element of N_0 to element of G_{x0}.

Let $g \in N_0$ be stored in the array of length 5 referred by parameter `g` as described in the documentation of file `mm_group_n.c`. We convert g to an element in **G_x0 representation** and store the result in the array `elem`.

The function returns 0 in case of success and `ERR_QSTATE12_GX0_BAD_ELEM` if g is not in G_{x0}.

int32_t **`xsp2co1_conjugate_elem`**(uint64_t *elem, uint32_t *a, uint32_t n)

Conjugate element of G_{x0} by an element of monster group.

Let $g \in G_{x0}$ be stored in the array `elem` in **G_x0 representation**. Let w be a word in the generators of the monster group of length `n`. w is stored in the array `a`, and each entry of `a` encodes a generator of the monster group described in file `mmgroup_generators.h`.

The function tries to replace g by $h = w^{-1} g w$. The function succeeds if for any prefix w_i of the word w we have $w_i^{-1} g w_i \in G_{x0}$.

int32_t **`xsp2co1_power_elem`**(uint64_t *elem1, int64_t e, uint64_t *elem2)

Exponentiation of an element of the group G_{x0}.

Let $g \in G_{x0}$ be stored in the array `elem1` in **G_x0 representation**. The function computes the power g^e and stores the result in the array `elem2` in **G_x0 representation**. Here $-2^{63} < e < 2^{63}$ must hold.

A negative return value indicates an error.

Any kind of overlapping beween the arrays `elem1` and `elem2` is allowed.

int32_t **`xsp2co1_power_word`**(uint32_t *a1, uint32_t n, int64_t e, uint32_t *a2)

Exponentiation of an word of the group G_{x0}.

Let w be a word in the generators of G_{x0} of length `n`. w is stored in the array `a1`, and each entry of `a1` encodes a generator of G_{x0} as described in file `mmgroup_generators.h`.

The function stores w^e in the array `a2` in the same format as the word w and returns the length of the computed word, which is at most 10.

A negative return value indicates an error.

Any kind of overlapping beween the arrays `a1` and `a2` is allowed.

int32_t **`xsp2co1_odd_order_bitmatrix`**(uint64_t *bm)

Compute odd part of the order of an element of Co_1.

Let an element g of Co_1 be given as a 24 times 24 bit matrix in the array `bm` acting on the Leech lattice modulo 2 by right multiplication. Here vectors in the Leech lattice modulo 2 are given in **Leech lattice encoding**.

Then the function returns the odd part of the order of g. It returns a negative value in case of failure.

int32_t **`xsp2co1_half_order_elem`**(uint64_t *elem1, uint64_t *elem2)

Compute (halved) order of an element of the group G_{x0}.

Let $g \in G_{x0}$ be stored in the array `elem1` in **G_x0 representation**. The function returns the order of the element g.

If the order o of g is even then the function stores $g^{o/2}$ in the array `elem2` in **G_x0 representation**. Otherwise it stores the neutral element in `elem2`.

A negative return value indicates an error.

Any kind of overlapping beween the arrays `elem1` and `elem2` is allowed.

int32_t **`xsp2co1_order_elem`**(uint64_t *elem)

Compute order of an element of the group G_{x0}.

Let $g \in G_{x0}$ be stored in the array `elem1` in **G_x0 representation**. The function returns the order of the element g.

A negative return value indicates an error.

int32_t **`xsp2co1_half_order_word`**(uint32_t *a1, uint32_t n, uint32_t *a2)

Compute (halved) order of a word of the group G_{x0}.

Let w be a word in the generators of G_{x0} of length `n`. w is stored in the array `a1`, and each entry of `a1` encodes a generator of G_{x0} as described in file `mmgroup_generators.h`.

If the order o of the word w is even then the function stores $w^{o/2}$ in the array `a2` in the same format as the word w. Otherwise it stores the empty word in `a2`. The length `k` of the word in `a2` is at most 10.

The function returns the value `0x100 *o + k`.

A negative return value indicates an error.

int32_t **xsp2co1_order_word**(uint32_t *a, uint32_t n)

Compute order of a word of the group G_{x0}.

Let w be a word in the generators of G_{x0} of length `n`. w is stored in the array `a`, and each entry of `a` encodes a generator of G_{x0} as described in file `mmgroup_generators.h`.

The function returns the order of w .

A negative return value indicates an error.

uint32_t **xsp2co1_leech2_count_type2**(uint64_t *a, uint32_t n)

Count type-2 vectors in an affine subspace of the Leech lattice mod 2.

This function returns the number of type-2 vectors in an affine subspace V of the Leech lattice mod 2. Subspace V is defined by an array a of length n of bit vectors. V is given by:

$V = \{a_0 + \sum_{i=1}^{n-1} \lambda_i a_i \mid \lambda_i = 0, 1\}$.

Caution:

The function may change the description of the affine space V in the array a to a different description of the same space V.

Remark:

This function is a much faster version of the function `gen_leech2_count_type2` in file `gen_leech.c`. The implementation of the latter function is much simpler; so we keep it for test purposes.

int32_t **xsp2co1_trace_98280**(uint64_t *elem, int32_t (*f_fast)(uint64_t*))

Compute character of ρ_{98280} of element of G_{x0}.

This function is for internal use only.

Let $g \in G_{x0}$ be stored in the array `elem` in **G_x0 representation**. The function returns the character of the representation ρ_{98280}.

This function may takes a long time is it does not use precomputed tables. However, precomputing such tables may requires this function (being alled without any precomputed tables).

In parameter `f_fast` the user may specify a function with signature `int32_t (*f_fast)(uint64_t *elem)` that returns the character ρ_{98280} in some cases. That function should use precomputed tables for computing that character and return an error code if it cannot compute a character. In this case we use the standard method form computing the requested character. If `f_fast` is `NULL` then we always use the standard method.

Any value less than -0x1000000 returned by function `f_fast` or by this function is to be interpreted as an error.

int32_t **xsp2co1_traces_small**(uint64_t *elem, int32_t *ptrace)

Workhorse for function `xsp2co1_traces_all`

This function is for internal use only.

Parameters and operation are are as in function `xsp2co1_traces_all`. But this function does not compute the character of the representation ρ_{98280} in `ptrace[3]`.

int32_t **xsp2co1_traces_all**(uint64_t *elem, int32_t *ptrace)

Compute relevant characters of element of G_{x0}.

This function is for internal purposes only.

Let $g \in G_{x0}$ be stored in the array `elem` in **G_x0 representation**. The function computes the characters of the representations $\rho_{24}, \rho_{576}, \rho_{4096}, \rho_{98280}$ and stores the result in `ptrace[0],...,` `ptrace[3]` in that order.

This function returns 0 in case of success and a nonzero value otherwise.

There is a considerably faster function `xsp2co1_traces_all` in module `xsp2co1_traces.c` performing exactly the same operation depending in the same input parameters. More details are given in the documentation of that function.

In contrast to function `xsp2co1_traces_fast`, this function does not use any precomputed tables. Actually, this function is used for precomputing those tables.

int32_t **`xsp2co1_rand_word_N_0`**(uint32_t *w, uint32_t in_N_x0, uint32_t even, uint64_t *seed)

Generate a random element of the group N_0.

The function computes a uniform distributed random element g of the subgroup N_0 of structure $2^{2+11+22}.(M_{24} \times \mathrm{Sym}_3)$ of the monster. The group N_0 is generated by the generators with tags `x, y, d, p, t`.

Thw function stores a word representing the element g in the buffer `w` and returns the length of that word.

The length of the word in the buffer `w` is at most 5.

If parameter `in_N_x0` is nonzero then we compute a random element of the subgroup N_{x0} of index 3 in N_0 generated by the generators with tags `x, y, d, p`.

If parameter `even` is nonzero then we compute a random element of the subgroup N_{even} of index 2 in N_0 generated by the generators with tags `x, y, d, p, t`, where all generators with tag `d` correspond to even Golay cocode words.

If both, `in_N_x0` and `even`, are nonzero then we compute a random element of $N_{xyz0} = N_{\mathrm{even}} \cap N_{x0}$.

The function uses the internal random generator in file `gen_random.c`. Parameter `seed` must be a seed for a random generator as described in file `gen_random.c`.

int32_t **`xsp2co1_rand_word_G_x0`**(uint32_t *w, uint64_t *seed)

Generate a random element of the group G_{x0}.

The function computes a uniform distributed random element g of the group G_{x0}. It stores a word representing the element g in the buffer `w` and returns the length of that word.

The length of the word in the buffer `w` is at most 10.

The function uses the internal random generator in file `gen_random.c`. Parameter `seed` must be a seed for a random generator as described in file `gen_random.c`.

A negative return value indicates an error.

3.4.12 C functions in `leech3matrix.c`

File `leech3matrix.c` contains functions for computing with matrices corresponding to the part with tag 'A' of a vector of the representation of the monster modulo 3. Note that this part has a natural interpretation as a symmetric matrix on the Leech lattice.

For these computations we deal with `i0` times `i1` matrices `m` for `i0 <= 24`, `i1 <= 48`, Such a matrix is stored in an array `a` of integers of type `uint64_t` of length 24 * 3. Here the entry `m[i,j]` is stored in `a[3*i + j/16]`, bits `4 * (j % 16),..., 4 * (j % 16) + 3`. We call `a` the **matrix mod 3** encoding of the matrix `m`.

Unless otherwise stated, we assume that the lower two bits of such a bit field have arbitrary values, and that the higher two bits of that bit field are zero.

There are functions for loading such a matrix `m` from a vector in a represenation of the monster, for elechonization of `m`, for computing the kernel of `m` etc.

Functions

void **`leech3matrix_echelon`**(uint64_t *a)

Echelonize a matrix of integers mod 3.

Here `a` is a matrix in **matrix mod 3** encoding as documented in the header of this file. That matrix is transformed to row echelon form. We echelonize columns 0,. . . ,23 of matrix `a` in that order. The matrix is not converted to reduced echelon form.

void **`leech3matrix_compress`**(uint64_t *a, uint64_t *v)

compress a matrix in *matrix mod 3* encoding

Let `a` be an 24 times 48 matrix in *matrix mod 3* encoding. We consider `a` as a pair of two matrices `Ah, Al`, with`Al` in columns 0,. . . ,23 and `Ah` in columns 24,. . . ,47 of `a`.

We store matrix `Al` in the entries`v[0], ..., v[23]`, and matrix `Ah` in the entries`v[24], ..., v[47]`. Here column `j` of a row of `Ah` or `Al` is reduced modulo 3 (so it has value 0, 1, or 2) and that value is stored in bits `2*j+1` and `2*j` of the corresponding entry of `v`.

So each of the matrices `Al` and `Ah` will be encoded as the part with tag 'A' of a vector in the representation ρ_3 of the monster modulo 3.

The overlapping `v == a` is legal; any other kind of overlappig between `v` and `a` is illegal.

void **`leech3matrix_sub_diag`**(uint64_t *a, uint64_t diag, uint32_t offset)

Subtract diagonal matrix from matrix in *matrix mod 3* encoding.

Let `a` be an 24 times 48 matrix in *matrix mod 3* encoding.

We subtract a diagonal matrix from `a`. More precisely, we subtract the integer `diag` from all entries `a[i, i+offset]`, for `i = 0,...,23`.

uint64_t **`leech3matrix_rank`**(uint64_t *a, uint32_t d)

Rank and kernel of a `24 times 24` matrix modulo 3.

Let `r` be the rank of the `24 times 24` matrix `b = a - d * 1`. Here the entries of that matrix are taken modulo 3, `d` is an integer, and `1` is the unit matrix. Input `a` is a 24 times 24 matrix in **matrix mod 3** encoding as documented in the header of this file.

Let `r` be the rank of matrix `b` with entries taken modulo 3. If matrix `b` has rank 23 then its kernel is one dimensional. In that case the kernel contains two nonzero vectors `+-w`, and we define `w` to be one of these vectors. Otherwise we let `w` be the zero vector.

The function returns the value `(r << 48) + w`, with `w` the vector defined above given in *Leech lattice mod 3 encoding* as described in *The C interface of the mmgroup project*.

Input `a` is destroyed.

uint64_t **leech3matrix_vmul**(uint64_t v, uint64_t *a)

Multiply vector with a `24 times 24` matrix modulo 3.

Input `a` is a 24 times 24 matrix encoded as the part with tag 'A' of a vector in the representation ρ_3 of the monster modulo 3. Input `v` is a vector of 24 integers modulo 3 encoded in **Leech lattice mod 3** encoding. The function computes the product $v \cdot a$ of the vector v and the matrix a and returns the result in **Leech lattice mod 3** encoding.

Vector v has 24 entries. If the upper k entries of v are zero then we access the first $24 - k$ rows of matrix a only. So the buffer referred by `a` must have length (24 - k) in this case.

int32_t **leech3matrix_prep_type4**(uint64_t *a, uint32_t n, uint64_t *w, uint64_t *seed)

Prepare subspace of Leech lattice mod 3 for finding type-4 vectors.

Let a be the subspace of the Leech lattice mod 3 spanned by the vectors in the array `a` of length `n`. Here each entry of the array `a` is encoded in the same way as a row of the part with tag 'A' of a vector in the representation ρ_3 of the Monster modulo 3. Usually, such a subspace of the Leech lattice mod 3 is computed by applying function `leech3matrix_echelon` to a (suitably modified) part with tag 'A' of a vector in the representation ρ_3.

The function computes an array of `2*n` random vectors taken from the space a and stores these `2*n` vectors in the array `w` in **Leech lattice mod 3** encoding. Half of the vectors in array `w` are obtained by left multiplying matrix a with a random upper triangular matrix; and the other half of these vectors is obtained by left multiplication with a random lower triangular matrix. Parameter `seed` points to a random generator for generating random matrices; see module `gen_random.c` for details.

Thus the vectors in array `w` span the space a; and we may obtain random vectors in a by performing random additions and subtractions of the entries of `w`. E.g. function `leech3matrix_prep_type4` generates random type-4 in the space space a using that method.

The function returns the size `2*n` of the array `w` in case of success and a negative value in case of failure. Any overlapping between the arrays `a` and `w` is allowed. On input, `1 <= n <= 12` must hold.

int32_t **leech3matrix_rand_type4**(uint64_t *w, uint32_t n, uint32_t trials, uint64_t *seed)

Random type-4 vector in subspace of Leech lattice mod 3.

The function tries to find a random type-4 vector in a subspace a of the Leech lattice mod 3 spanned by the vectors in the array `a` of length `n`. Here the entries of the array `w` should be given in **Leech lattice mod 3 encoding**, as e.g. returned by function `leech3matrix_prep_type4`. That function returns more vectors generating the space a than necessary in order to facilitate the generation of random vectors in a.

The function performs up to `trials` random additions or subtractions in the space a, until it finds a type-4 vector. If such a type-4 vector has been found then that vector is returned as a vector `v` in the Leech lattice mod 2 in **Leech lattice encoding**. Parameter `seed` points to a random generator for generating the required random data; see module `gen_random.c` for details.

If a type-4 vector `v` has been found then the function returns `(t << 4) + v`. Here `0 <= v < 0x1000000` is the vector found in the Leech lattice mod 2 in **Leech lattice encoding**; and `t` is the number of trials required to find `v`. In case `t > 127` we put `t = 127`.

If no type-4 vector has been found after `trials` trials then the function returns 0.

The function returns a negative value in case of failure; e.g. if the random generator has failed.

int32_t **leech2matrix_add_eqn**(uint64_t *m, uint32_t nrows, uint32_t ncols, uint64_t a)

Add an equation to a system of linear bit equations.

The idea behind this function is that an external process generates rows of a bit matrix with `ncols` columns, with `0 < ncols <= 32`. This function checks such a row `a` and accepts it, if it linearly independent of all previously accepted rows. Thus at most `ncols` rows can be accepted. The `nrows` already accepted rows are stored in the array `m`. The function returns `1` if row `a` is accepted and `0` otherwise. A negative return value indicates an error. The size of the array `m` should be at least `ncols`.

Let `A` be the `ncols` times `ncols` matrix of all accepted rows `a[i]`, `0 <= i < ncols`; and let `I` be the `ncols` time `ncols` unit matrix. We left multiply `A` with a matrix `T` such that `T * A = I`. Thus `T = A**(-1)`. Technically, we perform row operations on the matrix `A[:nrows]` containing the first `nrows` lines already accepted, such that `T * A[:rnows]` is in **reduced echelon form**. We also perform the same row operations on the unit matrix to obtain `T`. We store `T[:rnows]` in columns `0,...,ncols-1` of matrix `M` and `T*A[:rnows]` in columns `ncols,...,2*ncols-1` of matrix `M`.

One may use function `leech2matrix_solve_eqn` for solving a system of linear equations obtained in that way.

uint32_t **leech2matrix_solve_eqn**(uint32_t *m, uint32_t ncols, uint64_t v)

Solve a system of linear bit equations.

Let `A` be a nonsingular `ncols` times `ncols` bit matrix stored in the array `m` in the special form as described in function `leech2matrix_add_eqn`.

The function returns the solution `w` of the equation `w * A = v`.

Caution:

Here `m` is of of type `uint32_t *`, but the corresponding parameter in function `leech2matrix_add_eqn` is of type `uint64_t *`. This simplifies the use of this function in most pplications.

uint32_t **leech2_matrix_basis**(uint32_t *v2, uint32_t n, uint64_t *basis, uint32_t d)

Subspace generated by vectors of Leech lattice modulo 2.

Compute a basis of the subspace of the Leech lattice modulo 2 generated by the vectors `v2[0],...,v2[n-1]`.

The function returns the dimension `k` of that subspace and computes a basis of that subspace in `basis[i], 0 <= i < k`.

Here `d` must be an upper bound for the dimension `k`. If `k` is unknown, one should put `d = 24`.

Bits 23,…,0 of the output matrix are echelonized in a special way. Here the columns are processed in the order:

11, 22, 21, …, 13, 12, 10, 9, …, 1, 0, 23.

One of the advantages of this echelonization is that the vector Ω (encoded as 0x800000) will occur in the basis if it is in the subspace, and that there are many even vectors (i.e. vectors orthogonal to Ω) in the basis.

int32_t **leech2_matrix_orthogonal**(uint64_t *a, uint64_t *b, uint32_t k)

Compute standard orthogonal complement in Leech lattice mod 2.

Let $A = a_0 \ldots, a_{k-1}$ be a matrix of k vectors in the Leech lattice mod 2 stored in the array `a`. The function returns a basis $B = b_0 \ldots, b_{23}$ of the Leech lattice mod 2 in the array `b`, and it returns a number m such that the vectors $b_m \ldots, b_{23}$ are a basis of the orthogonal complement of the space generated by the row vectors of A.

If the vectors $(a_0 \ldots, a_{k-1})$ are linear independent then the function returns $m = k$, and vector $b_i, i < k$ is orthogonal to all vectors a_j with $j \neq i$.

The basis $B = b_0 \ldots, b_{23}$ is stored in the array `b`.

We require $k \leq 24$. The function returns $m \geq 0$ in case of success a negative value in case of failure.

uint32_t **leech2_matrix_radical**(uint32_t *v2, uint32_t n, uint64_t *basis, uint32_t d)

Radical of subspace generated by vectors of Leech lattice mod 2.

Compute the radical of the subspace of the Leech lattice modulo 2 generated by the vectors `v2[0], ...,v2[n-1]`. Here the radical is the intersection of the space generated by `v2[0],...,v2[n-1]` with the orthogonal complement of that space.

Input parameters `v2, n,` and `d` are as in function `leech2_matrix_basis`. A basis of the radical of the space is computed in `basis`. The basis is echelonized as in function `leech2_matrix_basis`. The function returns the dimension `k` of radical spanned by that basis.

uint32_t **leech2_matrix_expand**(uint64_t *basis, uint32_t dim, uint32_t *v2)

List vectors in a subspace of the Leech lattice modulo 2.

The function computes all `2**dim` vectors of the subspace `V` of the Leech lattice modulo 2 given by the basis

`basis[0], ..., basis[dim - 1]` .

These vectors are written into the array `v2`. The function

void **leech3_vect_mod3_to_signs**(uint64_t *v, uint64_t mult, uint32_t n, uint64_t *signs)

Map vector in rep of the Monster mod 3 to array of signs.

Let `v` be a part of a vector of the 198884-dimensional representation of the monster group modulo 3, which is organized as a `n` times `24` matrix of integers mod 3. Here `v` is the relevant part of a vector encoded as in the `mmgroup.mm_op` extension; see *The C interface of the mmgroup project*, section *Description of the mmgroup.mm extension* for details.

Let `mult` be a vector of 24 integers mod 3 encoded in the **Leech lattice mod 3** encoding.

The function computes the scalar product of each row of `v` with `mult` and stores the `n` signs of these products in the array `sign`. Here the signs are stored in natural order and encoded as in function `qstate12_to_signs` in module `qstate12io.c`. As usual, the integers 0, 1, and 2 (mod 3) are mapped to 0, '+', and '-', respectively.

Output array `signs` should have at least `(n + 31) >> 5` entries.

3.4.13 C functions in `involutions.c`

File `involutions.c` contains functions for transforming involutions of the subgroup G_{x0} (of structure $2^{1+24}.\mathrm{Co}_1$) of the monster.

We try to transform such involutions to a standard form via conjugation by elements of the monster group.

Functions

int32_t **`xsp2co1_involution_invariants`**(uint64_t *elem, uint64_t *invar)

Compute invariant spaces for an involution in G_{x0}.

Let g be the element of the group G_{x0} stored in the array given by parameter `elem`. Let Λ_2 be the Leech lattice mod 2, with vectors in Λ_2 coded in **Leech lattice encoding** as usual. Conjugation by g is a linear operation on Λ_2, since the vectors in Λ_2 correspond to the elements of the normal subgroup Q_{x0} of structure 2^{1+24} (modulo the centre of G_{x0}). Let $A = A(g)$ be the 24×24 bit matrix that performs this operation on Λ_2 by right multiplication. Put $A_1 = A - 1$, and let I_1 be the image of matrix A_1.

In this function we require that the image of g in the factor group Co_1 of G_{x0} has order 1 or 2; othereise the function fails. That condition is equivalent to $A^2 = 1$, and also to $A_1^2 = 0$. If this is the case then we have:

$(\ker A_1)^\perp = I_1 \subset \ker A_1 = (I_1)^\perp$.

Any element $v \in \ker A_1$ is invariant under g, and so the corresponding element in Q_{x0} is invariant up to sign. The elements of Q_{x0} invariant under g (modulo the center of Q_{x0}) form a subspace $(\ker A_1)^+$ of $\ker A_1$ of codimension 0 or 1. Let $(I_1)^+$ be the orthogonal complement of $(\ker A_1)^+$. Then I_1 has the same codimension in $(I_1)^+$. The purpose of this function is to compute a basis of the smaller of the two spaces $(I_1)^+$ or $\ker A_1$.

We compute an output matrix in the array `invar` and return the number `k` of rows of that matrix in case of success. We use the following column bits of the output matrix.

23,…,0: Basis vector v_i of I_1 or $(I_1)^+$

55,…,32: Preimage (under A_1) of basis vector v_i, undefined if $v_i \notin I_1$

27: Here a nonzero bit in row 0 indicates an error.

In bits 24,…,26 of the output matrix we return the following linear forms on the space spanned by basis vectors:

Case 1: $I_1 = (I_1)^+$ or $I_1 = \ker A_1$

Then we return a basis of I_1, and we have $k = \dim I_1 = \dim (I_1)^+ \in \{0, 8, 12\}$

Bit 26: 0

Bit 25: type of basis vector (modulo 2)

Bit 24: sign of basis vector in I_1

Case 2: $I_1 \neq (I_1)^+$ and $I_1 \neq \ker A_1$

Then we return a basis of $(I_1)^+$, and we have $k - 1 = \dim (I_1)^+ - 1 = \dim I_1 \in \{0, 8\}$.

Bit 26: 0 if and only if the basis vector is in I_1

Bit 25: 0

Bit 24: sign of the basis vector if the vector is in I_1, and type of the basis vector (mod 2) otherwise

Parameter `invar` must be an array of length 12. Zero lines are appeded to that array so that its length will be 12.

The function returns the dimension `k` of the computed basis, and a negative value in case of error. The return value ERR_QSTATE12_GX0_BAD_ELEM means that that the image of g in Co_1 has order greater than 2.

Bits 26,…,0 of the output matrix are echelonized in a special way. Here the columns are processed in the order:

26, 25, 24, 11, 22, 21, …, 13, 12, 10, 9, …, 1, 0, 23.

One of the advantages of this echelonization is that the vector Ω (encoded as 0x800000) will occur in the basis if it is in the subspace, and that there are many even vectors (i.e. vectors orthogonal to Ω) in the basis. Also, bits 26, 25, 24 may be nonzero at most in the first two columns of the output matrix.

int32_t **`xsp2co1_involution_orthogonal`**(uint64_t *invar, uint32_t col)

Compute some orthogonal complement for involution invariants.

Let g be an element of the group G_{x0} such that the image of g in Co_1 has order 1 or 2. For that element g, let A, A_1, and I_1 be as in function `xsp2co1_involution_invariants`.

In this function the input parameter `invar` must be equal to the output `invar` of function `xsp2co1_involution_invariants` applied to the element g.

There is a nondegenerate bilinear form $\langle\langle ., . \rangle\rangle$ on I_1 given by

$\langle\langle x, y \rangle\rangle = \langle \pi(x), y \rangle$,

where $\pi(x)$ is any preimage of x under A_1, and $\langle ., . \rangle$ is the scalar product on the Leech lattice modulo 2. The form $\langle\langle ., . \rangle\rangle$ is also called the **Wall parametrization**, see [Wal63]. If the image of g in Co_1 has order at most two then the Wall parametrization is a symmetric bilinear form.

The function computes the orthogonal complement v of a linear form l on the Leech lattice modulo 2 under the Wall parametrization. Then v is a vector in the Leech lattice modulo two. If parameter `col` is 0 or 1 then we let l be the linear form in column `col + 25` of matrix `invar`.

The function returns v in case of success and a negative value in case of failure.

[Wal63] G. E. Wall. On the conjugacy classes in the unitary, symplectic and orthogonal groups. J. Australian Math. Soc. 3, pp 1–63, 1963.

int32_t **`xsp2co1_involution_find_type4`**(uint64_t *invar, uint32_t guide)

Find type-4 vector in a space computed by `xsp2co1_involution_invariants`.

Let g be the element of the group G_{x0}, and for that element g let A, A_1, I_1, and $(I_1)^+$ be as in function `xsp2co1_involution_invariants`.

Here input parameter `invar` must be the output `invar` of function `xsp2co1_involution_invariants` applied to the element g. This function is successful in case $\dim I_1 = 8$ only.

We return a type-4 vector the space I_1. If no such vector exists then we return 0.

Parameter `guide` should usually be zero. If `guide` is a type-4 vector in the Leech lattice mod 2 satisfying the assumptions the return value vthen the function returns `v = guide`. Otherwise parameter `guide` is ignored.

int32_t **`xsp2co1_elem_find_type4`**(uint64_t *elem, uint32_t guide)

Try to simplify an element in G_{x0} via conjugation.

Let g be the element of the group G_{x0} stored in the array given by parameter `elem`. In this function we require that the image of g in the factor group Co_1 of G_{x0} has order 1 or 2; otherwise the function fails.

Let Λ_2 be the Leech lattice mod 2, with vectors in Λ_2 coded in **Leech lattice encoding** as usual. Let Ω be the standard frame in Λ_2.

Then the function tries to find a vector $v \in \Lambda_2$ with the following property:

For any $h \in G_{x0}$ with $v \cdot h = \Omega$ we have $h^{-1} g h \in N_{x0}$.

The function returns v in case of success and a negative value in case of an error. It returns `ERR_QSTATE12_GX0_BAD_ELEM` if no suitable vector v has been found.

The function succeeds if the following two conditions hold:

- g is in class 1A, 2A, 2B or 4A of the monster group
- g^2 is in subgroup Q_{x0} of G_{x0}.

In these two cases there is also a power τ^e of the triality element τ with

$\tau^{-e} h^{-1} g h \tau^e \in Q_{x0}$.

Caution:

The last statement has been checked for classes 1A, 2A and 2B only!

Parameter `guide` should usually be zero. If `guide` is a type-4 vector in the Leech lattice mod 2 satisfying the assumptions the return value `v`then the function returns `v = guide`. Otherwise parameter `guide` is ignored. It is also ignored in case $g \in Q_{x0}$.

int32_t **`xsp2co1_elem_conj_G_x0_to_Q_x0`**(uint64_t *elem, uint32_t *a)

Try to map an element of G_{x0} to Q_{x0}.

Let g be the element of the group G_{x0} stored in the array given by parameter `elem`. The function tries to find an element h in the monster group with $h^{-1} g h = q \in Q_{x0}$.

The function succeeds if the following two conditions hold:

- g is in class 1A, 2A, 2B or 4A of the monster group
- g^2 is in subgroup Q_{x0} of G_{x0}.

The function stores h in the output array `a` as a word of generators of the monster group. Array `a` must be of length `7` . The function returns q in bits `24,...,0` of the return value, and number of atoms in the array `a` in bits `27, 26, 25` of the return value The data in the array `a` are padded with zeros.

The function returns a negative value in case of failure. It returns `ERR_QSTATE12_GX0_BAD_ELEM` if no suitable element h can be found.

int32_t **`xsp2co1_elem_conjugate_involution`**(uint64_t *elem, uint32_t *a)

Map an involution in G_{x0} to a standard form.

Let g be an involution of the group G_{x0} stored in the array given by parameter `elem`.

The function computes an element a in the monster such that $h = a^{-1} g a$ is one of the following elements of the subgroup Q_{x0} of G_{x0}:

If $g = 1$ then $h = a = 1$.

If g is a 2A involution then h is the involution in Q_{x0} corresponding to the Golay cocode word with entries $2, 3$ being set.

If g is a 2B involution then h is the central involution z in Q_{x0}.

The element a is stored in the array `a` as a word of generators of the monster group. In case of success the function returns `0x100 * I + len(a)`, where `len(a)` is the length of the array `a`. We put `I = 0` if $g = 1$. We put `I = 1, 2` if g is a 2A or 2B involution, respectively.

The function returns `ERR_QSTATE12_GX0_BAD_ELEM` if g is not an involution.

The array `a` must have length at least 14.

3.4.14 C functions in `xsp2co1_traces.c`

File `xsp2co1_traces.c` contains functions for computing characters of some representations of the subgroup G_{x0} (of structure $2^{1+24}.\mathrm{Co}_1$) of the monster.

Such computations can be very expensive, especially for some classes of involutions, or for elements that map to involutions in the factor group Co_1 of G_{x0}.

This file contains a function `xsp2co1_elem_involution_class` for the classification of elements that map to involutions in Co_1.

Function `xsp2co1_traces_fast` uses a precomputed table for computing the characters of elements of G_{x0}. That table is addressed by the class information computed by function `xsp2co1_elem_involution_class`. The functions in module `mmgroup\tests\test_involutions.make_involution_samples.py` precompute that table. We simply copy and paste the table from the output of that python function to to this file.

The precomputation of the table requires the function `xsp2co1_traces_all` in file `xsp2co1_elem.c`. That function computes the same characters as function `xsp2co1_traces_fast` without using precomputed tables.

Function `xsp2co1_elem_involution_class` does not use a precomputed table, but the verification of this function requires inspection of the output of the module `make_involution_samples.py` mentioned above.

Functions

int32_t **`xsp2co1_elem_involution_class`**(uint64_t *elem)

Compute class information for certain elements of G_{x0}

Let $g \in G_{x0}$ be stored in the array `elem` in **G_x0 representation**. If g maps to an involution in the factor group Co_1 of G_{x0} then the function returns a nonzero value indicating some class information about g. Otherwise the function returns 0.

The class information in the return value is to interpreted as follows:

```
bits  7 .. 0: class of element  g  in the Monster group, e.g
              0x21 means class 2A,
              0x41 means class 4A, 0x42 means class 4B, etc.

bits 11 .. 8: Class of element  g  in the factor group Co_1
              0 means class 1A in Co_1
              1 means class 2A in Co_1
              2 means class 2B in Co_1
              3 means class 2C in Co_1

bit 12:       0 if  g  and  -g  are in the same class in the Monster
              1 otherwise

bit 13:       1 if q g  is equal to or powers up to -1
              0 otherwise
```

All other bits in the return value are set to zero.

Here -1 is the central involution x_{-1} in G_{x0}, and $-g = x_{-1} \cdot g$.

Write $h(g)$ as an abbreviation for the result of this function applied to an element g of G_{x0}. Then the following assertions have been checked computationally in files `make_involution_samples.py`, or `test_xp2_traces.py`, or can easily be checked mathematically.

Possible values $h(g)$ (depending on the class of gQ_{x0}) in Co_1 are:

```
class 1A: 0x1011, 0x3022, 0x0022, 0x0021, 0x2041
class 2A: 0x1121, 0x1122, 0x0143, 0x2143, 0x0142, 0x0141, 0x0122
class 2B: 0x0244, 0x2244
class 2C: 0x0322, 0x0341, 0x0344, 0x2382, 0x0343, 0x0342
```

The value $h(g)$ determines the characters of the representations $98280_x, 299_x, 24_x, 4096_x$ of g uniquely, where by construction of G_{x0} the last two characters are determined up to sign only.

The class of an involution g is determined uniquely by $h(g)$.

int32_t **`xsp2co1_traces_fast`**(uint64_t *elem, int32_t *ptrace)

Compute relevant characters of element of G_{x0}.

Let $g \in G_{x0}$ be stored in the array `elem` in **G_x0 representation**. The function computes the characters of the representations $\rho_{24}, \rho_{576}, \rho_{4096}, \rho_{98280}$ and stores the result in `ptrace[0],...,` `ptrace[3]` in that order. Here ρ_{576} is the tensor square of ρ_{24}.

This function returns 0 in case of success and a nonzero value otherwise.

Note that the tensor product $\rho_{24} \otimes \rho_{4096}$ is well defined, but the factors of that product are defined up to sign only. We normalize the characters corresponding to ρ_{24} and ρ_{4096} so that the first nonzero value of these two characters (in the order given above) is positive.

So this function performs the same action as function `xsp2co1_traces_all` in file `xsp2co1_elem.c`, but it is considerably faster, since it uses precomputed tables for som hard cases.

int32_t **`xsp2co1_elem_conjugate_involution_Gx0`**(uint64_t *elem, uint32_t guide, uint32_t *a)

Map an involution in G_{x0} to a standard form.

Let g be an involution in the group G_{x0} stored in the array given by parameter `elem` in **G_x0 representation**.

The function computes an element a in G_{x0} such that $h = a^{-1}ga$ is a (fixed) representative of the class of g in the group G_{x0}.

The element a is stored in the array `a` as a word of generators of the monster group. In case of success the function returns `0x100 * iclass + len(a)`, where `len(a)` is the length of the data in the array `a`, and `iclass` is explained below. The function returns a negative value in case of failure, e.g. if g has order greater than 2. The array `a` must have length at least 10.

In the sequel we list the representatives of all classes of involutions in G_{x0} computed by this function. For any such representative we also list the number `iclass` indicating the class of the involution as computed by function `xsp2co1_elem_involution_class`.

`iclass = 0x1101`: the neutral element x_1

`iclass = 0x3022`: the central involution x_{-1}

`iclass = 0x0021`: the element $x_{\{2,3\}}$

`iclass = 0x0022`: the element x_Ω

`iclass = 0x1121`: the element y_o

`iclass = 0x1122`: the element $x_{-1}y_o$

`iclass = 0x0122`: the element $y_o x_{\{8,9\}}$

`iclass = 0x0322`: the element $y_D x_{\{0,12\}}$

Here in $x_{\{i,j\}}$ the index $\{i, j\}$ indicates a Golay cocode word of length 2 given by the entries i and j. Octad o is the standard octad $\{0, 1, 2, 3, 4, 5, 6, 7\}$. Dodecad D is the standard dodecad $\{0, 4, 8, 13, 14, 15, 17, 18, 19, 21, 22, 23\}$.

Parameter `guide` should usually be zero. If `guide` is a type-4 vector v_4 in the Leech lattice mod 2 such that the two conditions $h = a^{-1}ga$ and $v_4 \cdot a = \Omega$ can both be achieved then we compute an element a satisfying these two conditions. Otherwise parameter `guide` is ignored. Here Ω is the standard frame in the Leech lattice.

int32_t **`xsp2co1_map_involution_class_Gx0`**(uint32_t iclass, uint32_t *a)

Map an involution class in G_{x0} to its representative.

Here parameter `class` must be a class number of an involution in the group G_{x0} as returned by function `xsp2co1_elem_conjugate_involution_Gx0`.

Then the function computes the representative h of the class of involutions in G_{x0} as it is computed by function `xsp2co1_elem_conjugate_involution_Gx0`.

The element h is stored in the array `a` as a word of generators of the monster group. In case of success the function returns the length `len(a)` of the data in the array `a`. The function returns a negative value in case of failure, e.g. if `iclass` does not correspond to an involution. The array `a` must have length at least 2.

3.4.15 C functions in `xsp2co1_map.c`

File `xsp2co1_map.c` contains functions for computing an element g of the group $G_{x0} = 2^{1+24}.\mathrm{Co}_1$ from the action of g on the normal subgroup $Q_{x0} = 2^{1+24}$ of G_{x0}.

Here the group G_{x0} is the maximal subgroup of the Monster used in our construction of the Monster. We store an element of G_{x0} as word of generators of that group as described in file`mmgroup_generators.h`. Internally, we also use the **G_x0 representation** for elements of G_{x0} as described in file `xsp2co1.c`.

Elements of the group Q_{x0} are stored in **Leech lattice encoding** as described in section **Description of the mmgroup.generators extension**.

Note that an element of G_{x0} is determined by its action on Q_{x0} up to sign only.

The main function `xsp2co1_elem_from_mapping` in this module tries to find the 'nicer' of the elements $\pm g$ from its action on Q_{x0}.

Functions

int32_t **`xsp2co1_Co1_get_mapping`**(uint32_t *m1, uint32_t *m2, uint32_t *m_out)

Compute a certain mapping from Q_{x0} to itself.

Let $g \in G_{x0}$ be such that g maps $m_{1,j}$ to $m_{2,j}$ via conjugation, for $m_{i,j} \in Q_{x0}, i = 1, 2; 0 \leq j \leq 24$. If the $m_{1,j}$ (considered as vectors in $\Lambda/2\Lambda$) are linear independent then there is at most one such g, up to sign.

Here inputs $m_{1,j}, m_{2,j}$ are given in the arrays `m1, m2` in **Leech lattice encoding**.

The function computes g as a mapping $m_{0,j} \mapsto m_{3,j}$, where $m_{0,j}$ is the standard basis of $\Lambda/2\Lambda$ (with $m_{0,j}$ `= 1 << j` in **Leech lattice encoding**). The function stores the vectors $m_{3,j}$ in the array `m_out` of length 24 in **Leech lattice encoding**.

Let o be the odd part of the order of g, so that g has order $2^k \cdot o$.

The function returns a negative value if it detects an error, and it returns o if it does not detect any error. If the function returns $o \geq 0$ and the output is a correct image of the standard basis then there exists a $g \in G_{x0}$ that maps $m_{1,j}$ to $m_{2,j}$. The order of any such element g divided by o is a power of two.

Any overlapping between the arrays referred by `m1, m2, m_out` is allowed.

int32_t **`xsp2co1_Co1_matrix_to_word`**(uint32_t *m, uint32_t *g)

Compute preimage in G_{x0} of automorphism on Q_{x0}.

Let matrix m (given by parameter `m`) be a 24 times 25 bit matrix that describes an automorphism g' acting on Q_{x0}. Here row `m[i]` is the image of the (positive) element in Q_{x0} corresponding to the `i`-th basis vector of $\Lambda/2\Lambda$, and `m[i]` is encoded in **Leech lattice encoding**.

If possible, the function computes a $g \in G_{x0}$ that acts on Q_{x0} by conjugation in the same way as g' acts on Q_{x0}. If such a g exists, it is determined up to sign only.

In case of success the function stores g as a word of generators of G_{x0} in the buffer referred by parameter `g` and returns the length of that word. In case of failure the function returns a negative value.

Array `g` must have length at least 10.

int32_t **`xsp2co1_elem_from_mapping`**(uint32_t *m1, uint32_t *m2, uint32_t *g)

Compute $g \in G_{x0}$ from its operation on Q_{x0}.

Let $g \in G_{x0}$ be such that g maps $m_{1,j}$ to $m_{2,j}$ via conjugation, for $m_{i,j} \in Q_{x0}, i = 1, 2; 0 \leq j \leq 24$. If the $m_{1,j}$ (considered as vectors in $\Lambda/2\Lambda$) are linear independent then there is at most one such g, up to sign.

Here inputs $m_{1,j}, m_{2,j}$ are given in the arrays `m1, m2` in **Leech lattice encoding**.

If possible then the function computes a $g \in G_{x0}$ that maps $m_{1,j}$ to $m_{2,j}$ via conjugation. In case of success it stores g as a word of generators of G_{x0} in the buffer referred by parameter `g` and returns the length of that word in the lower 8 bits of the return value. In case of failure the function returns a negative value.

Array `g` must have length at least 10.

Note that g is determined up to sign only. The function makes a considerable effort to disambiguate the two elements $\pm g$.

If one of the two element $\pm g$ has odd order then the other one has necessarily even order; in that case we return the element with odd order. Otherwise both elements have the same (even) order $2^k \cdot o, o$ odd. Then at most one of the elements $\pm g^o$ may have a negative character $\chi(g^o)$ in the representation $\rho_{24} \otimes \rho_{4096}$ of G_{x0}; and in this case we return an element with $\chi(g^o) \geq 0$. This leads to a disambiguation of $\pm g$ in case $\chi(g^o) \neq 0$.

We store the order of the computed element g in bits 15…,8 of the return value. We set bit 16 of the return value precisely if we could disambiguate g from $-g$, i.e. in case $\chi(g^o) \neq 0$. (Note that $g^o = 1$ if g has odd order, implying $\chi(g^o) > 0$.)

The function returns a negative value in case of failure.

3.5 Description of the `mmgroup.mm_op` extension

Module `mmgroup.mm_op` is implemented as an extension in the `mmgroup` package implemented in `Cython`. The main source file for that extension is `mm_op.pyx` in directory `src.mmgroup.dev.mm_op`. Each documented `.c` in this module function has been wrapped by a `Cython` function with the same name and the same signature.

Module `mmgroup.mm_op` implements the 196884-dimensional rational representation ρ_p of the monster group modulo several fixed odd moduli $p = 2^k - 1$.

3.5.1 Representation of a vector in ρ_p

We describe the representations of a vector in ρ_p

The most important representation of a vector in ρ_p is the *internal* representation. All operations of the monster group are performed on vectors in in ρ_p in internal representation.

In the *internal* representation a vector is stored as an array of `247488` entries, where each entry is a bit field representing an integer modulo `p`. Some components of a vector are stored twice in that array and some entries of the array are unused. This special structure facilitates the implementation of the operations of the monster group.

The entries of a vector are organized as tuples `(tag, i0, i1)`. Here indices `i0, i1` refer to a two-dimensional array as indicated in column `Size` of the following table. As usual in C, entries with adjacent last index `i1` are stored in adjacent locations. For a mathematical description of an entry `(tag, i0, i1)`, see section *The Representation of the Monster Group* in the API reference.

Table 1: Array sizes for tags in the internal representation

Tag	Size	Space used	Remarks	Offset
`A`	`24 x 24`	`24 x 32`	(1), (2)	`0`
`B`	`24 x 24`	`24 x 32`	(1), (2), (3)	`768`
`C`	`24 x 24`	`24 x 32`	(1), (2), (3)	`1536`
`T`	`759 x 64`	`759 x 64`	(4),	`2304`
`X`	`2048 x 24`	`2048 x 32`	(1),	`50880`
`Z`	`2048 x 24`	`2048 x 32`	(1),	`116416`
`Y`	`2048 x 24`	`2048 x 32`	(1),	`181952`

Remarks

1. As indicated in column `Space used`, an array of size `n` times `24` is stored in a space reserved for an array of size `n` times `32`. So the entry with index `(tag, i0, i1)` is stored at location `Offset(tag) + 32 * i0 + i1`. Unused entries must be equal to zero.
2. The entry given by `(tag, i0, i1)` must be equal to the entry given by `(tag, i1, i0)`.
3. The diagonal entry given by `(tag, i0, i0)` must be equal to zero.
4. The entry with index `(T, i0, i1)` is stored at location `Offset(T) + 64 * i0 + i1`.

The entries of a vector in internal representation are stored a one-dimensional array of integers of type `uint_mmv_t`. Here type `uint_mmv_t` may be one of the C integer types `uint64_t` or `uint32_t`, depending on the value `INT_BITS`. At present `INT_BITS` is set to the value 64. Several adjacent entries of a vector are stored as bit fields in a single integer of type `uint_mmv_t`. Entries with a lower index are stored at bits with lower valence.

The number of bits in a bit field is always a power of two. So e.g. for `p = 3` we use 2 bits; for `p = 7` we use 4 bits with the highest bit unused. In case `p = 2**k - 1`, legal values for an entry are `0,...,2**k - 1`, with `2**k - 1` equal to `0`. Thus negation of a value can be done by complementing all `k` bits of that value. Apart from negation, the matrices corresponding to operations of the monster may add, subtract and half the entries of a vector (modulo p). These operations can easily be done on several entries simultaneously by manipulating a just single integer of type uint_mmv_t.

As indicated above, we reserve 32 entries for arrays of integers modulo p with 24 entries. So we require about 25.7% more memory than necessary. In some cases we need this extra memory anyway. E.g. for p = 3 a 64-bit integer may store 32 entries, so that there will always be a slack of 8 entries when storing 24 entries.

Function mm_aux_mmv_size(p) returns the number of integers of type uint_mmv_t required for storing a vector in external representation.

When writing or calculating entries with tags A, B, C then the high-level function for manipulating vectors in internal representation make sure that e.g. entries with indices (A, i0, i1) and (A, i1, i0) are always set to the same value. These functions also make sure that unused bits in a bit field and unused bit fields are set to zero. Note that a bit field with value zero may contain the value p instead.

The *external* representation of a vector in ρ_p

There is also a so-called *external representation* of a vector in R_p. This is used to facilitate the access to vectors by external modules. Here the vector is represented as an array of 196884 integers of type uint8_t. Basis vectors are ordered similar to the ordering for the internal representation, but here the entries are in one-to-one correspondence with the basis vectors. In the external representation there are no unused or duplicated entries.

More precisely, the order of the entries is:

Table 2: Order of entries in external representation

Entries	Condition	No of entries	Offset
(A, i0, i1)	i0 = i1	24	0
(A, i0, i1)	i0 > i1	276	24
(B, i0, i1)	i0 > i1	276	300
(C, i0, i1)	i0 > i1	276	576
(T, i0, i1)		759*64	852
(X, i0, i1)		2048*24	49428
(Z, i0, i1)		2048*24	98580
(Y, i0, i1)		2048*24	147732

Indices (tag, i,j) for tag = A, B, C, i > j are ordered as follows:

- (1,0),
- (2,0), (2,1),
- (3,0), (3,1), (3,2),
- ...
- (i,0), (i,1), ..., (i,i-1),
- ...
- (24,0), (24,1), ..., (24,23).

Function mm_aux_bytes_to_mmv() converts a vector from external to internal representation, Function mm_aux_mmv_to_bytes() does the inverse conversion.

The *sparse* representation of a vector in ρ_p

The Python interface to vectors in ρ_p is optimized for readability and not for speed. Here a typical task is to read and modify single entries of a vector. In the internal representation the coordinates of a vector are indexed by tuples containing a string. Transferring tuples or strings from Python to C is awful. Here we need a representation of a vector in ρ_p where a single entry of a vector can be stored in an integer variable.

A vector in ρ_p can be stored in the *sparse* representation. Here a vector is stored as an array of 32-bit integers, where each entry stands for a multiple of a basis vector. A component of a vector is stored in the bit fields of an integer as a tuple `(tag, i0, i1, value)`. Here the tuple `(tag, i0, i1)` is as in the external representation, and `value` is the value of the coordinate of the vector corresponding to `(tag, i0, i1)`. Entries with coordinate zero may be dropped. A 32-bit integer encodes a tuple `(tag, i0, i1, value)` in bit fields as shown in the following table.

Table 3: Bit fields in the sparse representation

Bits	Meaning
27..25	Tag: `A = 1`, `B = 2`, `C = 3`, `T = 4`, `X = 5`, `Z = 6`, `Y = 7`
24..15	Index `i0`
13.. 8	Index `i1`
7.. 0	The `value` of the coordinate of the basis vector; if the modulus `p` is `2**k - 1` then only the lowest `k` bits are evaluated.

In a C function the length of a sparse representation of a vector must be given as a parameter to the function. The order of the entries is irrelevant in the sparse representation. A sparse representation generated by a C function contains at most one entry for each tuple `(tag, i0, i1)`. On input, entries with several equal tuples `(tag, i0, i1)` are accepted. Unless stated otherwise, the corresponding values of such equal tuples are added.

In an entry with tag `A`, `B`, or `C` generated by this module we always have `i0 >= i1`. The `value` of an entry generated by this module is always less than the modulus `p`.

When reading an entry, a `value` with `0 <= value <= p` is accepted. Entries with tag `A`, `B`, or `C` and `i < j` are also accepted. Illegal tags or indices are usually ignored on input.

3.5.2 Header file mm_basics.h

The header file `mm_basics.h` contains basic definitions for dealing with vectors of the 198884-dimensional representation of the monster group, as described in *The C interface of the mmgroup project*, section *Description of the mmgroup.mm extension*.

It also contains prototypes for the C files in the `mm` extension. This extension comprises the files `mm_aux.c`, `mm_crt.c`, `mm_group_word.c`, `mm_random.c`, `mm_tables.c`, `mm_tables_xi.c`.

Defines

`mm_aux_bad_p`(p)

Return 0 if `p` is a good modulus and a nonzero value otherwise.

Typedefs

typedef uint64_t **`uint_mmv_t`**

Used for the representation of the monster group.

Internally, a vector in the 196884-dimensional representation of the monster is stored as an array of integers of type `uint_mmv_t`. Here several entries are stored in such an integer. See `enum MM_AUX_OFS_type` for more details.

Enums

enum **`MM_AUX_OFS`**

This enumeration contains the offsets for the tags `A,B,C,T,X,Z,Y` in a vector in the 196884-dimensional representation of the monster, stored in the internal representation.

Such an offset counts the number of entries starting at the beginning of th vector. Note that several entries of a vector are stored in a 64-bit integer. Also there may be duplicate or unused entries in a vector, in order to speed up the operation of the monster group on a vector.

Values:

enumerator **`MM_AUX_OFS_A`**

Offset for tag A

enumerator **`MM_AUX_OFS_B`**

Offset for tag B

enumerator **`MM_AUX_OFS_C`**

Offset for tag C

enumerator **`MM_AUX_OFS_T`**

Offset for tag T

enumerator **`MM_AUX_OFS_X`**

Offset for tag X

enumerator **`MM_AUX_OFS_Z`**

Offset for tag Z

enumerator **`MM_AUX_OFS_Y`**

Offset for tag Y

enumerator **`MM_AUX_LEN_V`**

Total length of the internal representation

enum **`MM_AUX_XOFS`**

This enumeration contains the offsets for the tags `A,B,C,T,X,Z,Y` in a vector in the 196884-dimensional representation of the monster, stored in the external representation.

In external representation, a vector is stored as a contiguous array of bytes.

Values:

enumerator **`MM_AUX_XOFS_D`**

Offset for diagonal entries of tag A

enumerator **`MM_AUX_XOFS_A`**

Offset for tag A

enumerator **MM_AUX_XOFS_B**

Offset for tag B

enumerator **MM_AUX_XOFS_C**

Offset for tag C

enumerator **MM_AUX_XOFS_T**

Offset for tag T

enumerator **MM_AUX_XOFS_X**

Offset for tag X

enumerator **MM_AUX_XOFS_Z**

Offset for tag Z

enumerator **MM_AUX_XOFS_Y**

Offset for tag Y

enumerator **MM_AUX_XLEN_V**

Total length of the external representation

enum **MM_SPACE_TAG**

This enumeration defines the values of the tags `A,B,C,T,X,Z,Y` in a vector in the 196884-dimensional representation of the monster, stored in the sparse representation.

In the sparse representation an entry of a vector is stored as a tuple of bit fields `(tag, par1, par2, value)` inside an integer of type `uint32_t` as follows:

```
Bits 27,...,25:  tag (as indicated below)

Bits 24,...,14:  par1 (an integer of up to 11 bits)

Bits 13,..., 8:  par2 (an integer of up to 6 bits)

Bits  7,..., 0:  value (Reserved for the value of an entry)
```

Values:

enumerator **MM_SPACE_TAG_A**

Encodes tag A

enumerator **MM_SPACE_TAG_B**

Encodes tag B

enumerator **MM_SPACE_TAG_C**

Encodes tag C

enumerator **MM_SPACE_TAG_T**

Encodes tag T

enumerator **MM_SPACE_TAG_X**

Encodes tag X

enumerator **MM_SPACE_TAG_Z**

Encodes tag Z

enumerator **MM_SPACE_TAG_Y**

Encodes tag Y

struct **mm_sub_op_pi64_type**

#include "mm_basics.h" Auxiliary structure for the structure *mm_sub_op_pi_type*

An array of type *mm_sub_op_pi64_type*[759] encodes the operation of $x_\epsilon x_\pi$ on the representation of the monster group for entries with tag T. Assume that entry (T, i, j) is mapped to entry +-(T, i1, j1). Then i1 depends on i only, and j1 depends on i and j. For fixed i the mapping j -> j1 is linear if we consider the binary numbers j and j1 as bit vectors.

Entry i1 of the array of type *mm_sub_op_pi64_type*[759] describes the preimage of (T, i1, j1) for all 0 <= j1 < 64 as documented in the description of the members preimage and perm.

Note that the values 1, 3, 7, 15, 31, 63 occur as differences j1 ^ (j1 - 1) when counting j1 from 0 up to 63. So the preimage of (T, i1, j1) can be computed from the preimage of (T, i1, j1 - 1) using linearity and the approprate entry in member perm.

We remark that in case of an odd value epsilon the mapping for tag T requires a postprocessing step that cannot be derived from the infomration in this structure. Then entry (T, i, j) has to be negated if the bit weight of the subset of octade i corresponding to index j has bit weight 2 modulo 4.

In the sequel we describe the meaning of entry i1 an an array of elements of type *mm_sub_op_pi64_type*.

Public Members

uint16_t **preimage**

Bits 9…0 : preimage i such that (T, i, .) maps to +-(T, i1, .)

Bit 12: sign bit: (T, i, .) maps to -(T, i1, .) if bit 12 is set

uint8_t **perm**[6]

Member perm[k] is a value vsuch that (T, i, v) maps to +-(T, i1, 2 * 2**k - 1)

struct **mm_sub_op_pi_type**

#include "mm_basics.h" Structure used for preparing an operation $x_\epsilon x_\pi$.

Function mm_sub_prep_pi computes some tables required for the operation of $x_\epsilon x_\pi$ on the representation of the monster group, and stores these tables in a structure of type *mm_sub_op_pi_type*.

The structure of type *mm_sub_op_pi_type* has the following members:

Public Members

uint32_t **eps**

A 12-bit integer describing an element ϵ of the Golay cocode.

uint32_t **pi**

An integer describing the element π of the Mathieu group M_{24} as in module `mat24_functions.c`.

uint8_t **perm**[24]

The permutation `0...23 -> 0...23` given by the element π of M_{24}.

uint8_t **inv_perm**[24]

The inverse of the permutation `perm`.

uint32_t **benes_net**[9]

A representation of Benes network for computing permutationperm, as described in function `mat24_perm_to_net` in file `mat24_functions.c`.

uint16_t **tbl_perm24_big**[2048 + 72]

For tags `A, B, C, X, Y, Z`, an entry `(tag, i, j)` of the representation of the monster is mapped to entry `(tag1, i1, j1)`, with `i1` depending on `i` (and the tag), and `j1` depending on `j` only.

If `tbl_perm24_big[i1] & 0x7ff = i` for `0 <= i1 < 2048` then `(tag, i, j)` ia mapped to `(Tag, i1, perm[j])`, up to sign, for tags `X`, `Y` and `Z`. In case of an odd ϵ, tags `Y` and `Z` have to be exchanged. The value `tbl_perm24_big[2048 + 24*k + i1] & 0x7ff` describes the preimage of `(tag, i1, j1)` in a similar way, where `tag = A, B, C`, for `k = 0, 1, 2`.

Bits 12,…,15 of `tbl_perm24_big[i1]` encode the signs of the preimages of the corresponding entry of the rep. Bits 12, 13, and 14 refer to the signs for the preimages for the tags `X`, `Z` and `Y`, respectively. Bit 15 refers to the signs for the preimages for tags `A`, `B` and `C`. If the corresponding bit is set, the preimage has to be negated.

Note that function `mat24_op_all_autpl` in module `mat24_functions.c computes`the first 2048 entries of the table.

mm_sub_op_pi64_type ***tbl_perm64**

A description of the operation of $x_\epsilon x_\pi$ on the entries with tag `T`, see structure *`mm_sub_op_pi64_type`*. Entry `d` of the Arrary refers to the octad `o(d)` with number `d`. It contains the followint information_

Bits 5,…,0: Associator `\delta' = A(o(d), f))` encoded as a suboctad

Bits 11,…,6: Associator `a = A(o(d), ef))` encoded as a suboctad.

Caution:

Pointer `tbl_perm64` must be initialized with an array of type *`mm_sub_op_pi64_type`* `a_tbl_perm64[759]`.

struct **mm_sub_op_xy_type**

#include "mm_basics.h" Structure used for preparing an operation $y_f x_e x_\epsilon$.

The operation of $g = y_f x_e x_\epsilon$, (or, more precisely, of its inverse g^{-1}) on the representation of the monster group is described in section **Implementing generators of the Monster group** in the **The mmgroup guide for developers**.

Function `mm_sub_prep_xy` in file `mm_tables.c` collects the data required for this operation in a structure of type *`mm_sub_op_xy_type`*.

Public Members

uint32_t **`f`**

A 13-bit integer describing an element f of the Parker loop.

uint32_t **`e`**

A 13-bit integer describing an element e of the Parker loop.

uint32_t **`eps`**

A 12-bit integer describing an element ϵ of the Golay cocode.

uint32_t **`f_i`**

Bit i of member `f_i` is the scalar product of f and the singleton cocode word (i).

These bits are used for the operation of g^{-1} on entries with tag `A`.

uint32_t **`ef_i`**

Bit i of member `ef_i` is the scalar product of ef and the singleton cocode word (i).

These bits are used for the operation of g^{-1} on entries with tags `B`, and `C`.

uint32_t **`lin_i`**[3]

Put $g_0 = e$, $g_1 = g_2 = f$. For `k = 0, 1,2`, the bit i of member `lin_i[k]` is the scalar product of g_k and the singleton cocode word (i).

These bits are used for the operation of g^{-1} on entries with tags `X`, `Z`, and `Y`.

uint32_t **`lin_d`**[3]

Let `U_k = X, Z, Y` for `k = 0, 1, 2`. If the cocode element ϵ is even then we put `U'_k = U_k`, otherwise we put `U'_k = X, Y, Z` for `k = 0, 1, 2`. The operation g^{-1} maps the vector with tag `(U_k, d, i)` to `(-1)**s` times the vector with tag `(U'_k, d ^ lin[d], i)`. Here `**` denotes exponentiation and we have

`s` = `s(k, d, i)` = `(lin_i[k] >> i) + (sign_XYZ[d] >> k)`.

If `k = 0` and ϵ is odd then we have to correct `s(k, d, i)` by a term `<d, i>`.

uint8_t ***`sign_XYZ`**

Pointer `sign_XYZ` refers to an array of length 2048. This is used for calculations of signs as described above. Here we use the formula in section **Implementing generators of the Monster group** of the **mmgroup guide for developers**, dropping all terms depending on `i`.

uint16_t ***`s_T`**

Pointer `s_T` refers to an array of length 759. Entry `d` of this array refers to the octad `o(d)` with number `d`. The bits of entry `d` are interpreted as follows:

Bits 5,…,0: The asscociator `delta' = A(o(d), f)` encoded as a suboctad of octad `o(d))`.

Bits 13,…,8: The asscociator `alpha = A(o(d), ef)` encoded as a suboctad of octad `o(d))`. From his information we can compute the scalar product `<ef, \delta>` for each suboctad `delta` of `o(d)` as an intersection of tow suboctads. Here we assume that `delta` is represented as such a suboctad.

Bit 14: The sign bit `s(d) = P(d) + P(de) + <d, eps>`, where `P(.)` is the squaring map in the Parker loop.

Bit 15: Parity bit `|eps|` of the cocode word `eps`.

Then g^{-1} maps the vector with tag `(T, d, delta)` to `(-1)**s'` times the vector with tag `(T, d, \ delta ^ delta')`. Here `**` denotes exponentiation and we have

`s'` = `s'(T, d, delta)` = `s(d)` + `<\alpha, \delta>` + `|delta| * |eps| / 2`.

Here the product `<\alpha, \delta>` must be computed as the bit length of an intersection of two suboctads.

3.5.3 Header file mm_op_p.h

3.5.4 C interface for file mm_index.c

File `mm_index.c` provides the basic functions for converting an index of a vector of the 196884-diemnsional representation of the monster between internal, external, and sparse notation.

Functions

uint32_t **mm_aux_index_extern_to_sparse**(uint32_t i)

Convert an index from external to sparse representation.

The function converts an index `i` for the external representation of a vector to an index for the sparse representation of a vector and returns the converted index. The function returns 0 in case `i >= 196884``.

Indices for the sparse representation are defined as in `enum MM_SPACE_TAG` in file `mm_basics.h`.

void **mm_aux_array_extern_to_sparse**(uint32_t *a, uint32_t len)

Convert index array from external to sparse representation.

The function converts an array `a` of indices for the external representation to an array of indices for the sparse representation of a vector. All indices in the array `a` of length `len` are converted in place, using function `mm_aux_index_extern_to_sparse`.

int32_t **mm_aux_index_sparse_to_extern**(uint32_t i)

Convert an index from sparse to external representation.

The function converts an index `i` for the sparse representation of a vector to an index for the external representation of a vector and returns the converted index. The function returns -1 if the input `i` denotes an illegal index. The coordinate value encoded in the input `i` is ignored.

Indices for the sparse representation are defined as in `enum MM_SPACE_TAG` in file `mm_basics.h`.

int32_t **mm_aux_index_sparse_to_leech**(uint32_t i, int32_t *v)

Convert sparse index to a short vector in the Leech lattice.

The function converts an index `i` for the sparse representation of a vector to a vector `v` in the Leech lattice. This conversion is successful if `i` denotes a legal index for one of the tags tags `B, C, T, X`. Then the function computes a short Leech lattice vector (scaled to norm 32) in the array `v`. Output `v` is determined up to sign only; that sign is implementation dependent.

The function returns 0 in case of a successful conversion and -1 in case of failure.

uint32_t **mm_aux_index_sparse_to_leech2**(uint32_t i)

Convert sparse index to a short vector in the Leech lattice mod 2.

The function converts an index `i` for the sparse representation of a vector to a vector `v` in the Leech lattice mod 2. This conversion is successful if `i` denotes a legal index for one of the tags tags `B, C, T, X`. The function returns a short Leech lattice vector modulo 2, encoded in **Leech lattice encoding**, as described in section **Description of the mmgroup.generators extension**.

The function returns 0 in case of failure.

uint32_t **mm_aux_index_leech2_to_sparse**(uint32_t v2)

Convert short vector in the Leech lattice mod 2 to sparse rep.

The function converts an value `v2` representing a vector in the Leech lattice mod 2 to a sparse index and returns that sparse index. It returns 0 if `v2` is not a short Leech lattice vector.

uint32_t **mm_aux_index_intern_to_sparse**(uint32_t i)

Convert an index from internal to sparse representation.

The function converts an index `i` for the internal representation of a vector to an index for the sparse representation of a vector and returns the converted index. The function returns 0 in case of a bad index.

Indices for the sparse representation are defined as in `enum MM_SPACE_TAG` in file `mm_basics.h`.

int32_t **mm_aux_index_sparse_to_intern**(uint32_t i)

Convert an index from sparse to internal representation.

The function converts an index `i` for the sparse representation of a vector to an index for the internal representation of a vector and returns the converted index. The function returns -1 if the input `i` denotes an illegal index. The coordinate value encoded in the input `i` is ignored.

Indices for the sparse representation are defined as in `enum MM_SPACE_TAG` in file `mm_basics.h`.

int32_t **mm_aux_index_extern_to_intern**(uint32_t i)

Convert an index from external to internal representation.

The function converts an index `i` for the external representation of a vector to an index for the internal representation of a vector and returns the converted index. The function returns -1 in case `i >= 196884`.

int32_t **mm_aux_index_check_intern**(uint32_t i)

Check an index in internal representation.

The function checks an index `i` in the internal representation of a vector. Some entries of the vectors are stored at two different locations, e.g entries `A[i,j], B[i,j], C[i,j]` for `i != j`.

The function returns the other location of the same entry (as an index in internal representation) if there is any. It returns 0 if that entry is stored at exactly one location, and -1 if index `i` is illegal.

3.5.5 C interface for file mm_aux.c

File `mm_aux.c` provides the basic functions for dealing with the representations of the monster group modulo various small integers `p = 2**n-1`, `2 <= n <= 8`. Here the integer `p` is called the modulus.

Especially, we deal with vectors in such a representation as described in *The C interface of the mmgroup project*, section *Description of the mmgroup.mm extension*.

For such a vector there is an internal representation, an external representation, and also a sparse representation, as described in the documentation mentioned above.

The functions in this file provide access to the internal representation of such a vector. The also support the conversion between the different representations of a vector.

Usually, the order of the parameters of functions in this file is:

```
1. Modulus p, if present

2. The input value or the input data array

3. Any parameters that do not affect the positions in the output array

4. The output data array

5. Parameters (e.g. lengths, indices) that affect the positions of the
   data being modified in the output array
```

Among others, functions in this file use functions from file `mm_index.c` for converting indices of vectors to different representations.

A vector modulo `p` is organized in rows of 32 entries. In many rows only 24 of the 32 bits are used; but in some all 32 bit are used. Dtails are given in the *API reference* of the project.

Functions

uint8_t **`mm_aux_get_mmv`**(uint32_t p, *uint_mmv_t* *mv, uint32_t i)

Read entry at index from vector in internal representation.

The function returns the entry with index `i` of the vector `mv` with modulus `p`. The return value is reduced modulo `p`. Index `i` must be given in internal representation. The function returns garbage in case of an illegal index.

void **`mm_aux_put_mmv`**(uint32_t p, uint8_t value, *uint_mmv_t* *mv, uint32_t i)

Write entry to a vector at an index in internal representation.

The function sets the entry of the vector `mv` with modulus `p` at the index `i` to the given value. `0 <= value <= p` must hold.

Here the index `i` must be given in internal representation. Writing at an illegal index performs no action.

void **`mm_aux_add_mmv`**(uint32_t p, uint8_t value, *uint_mmv_t* *mv, uint32_t i)

Add to entry of a vector at an index in internal representation.

The function adds the given value to the entry of the vector `mv` with modulus `p` at the index `i`. Here `0 <= value <= p` must hold.

The index `i` must be given in internal representation. Writing at an illegal index performs no action.

void **`mm_aux_read_mmv32`**(uint32_t p, *uint_mmv_t* *mv, uint32_t i, uint8_t *b, uint32_t len)

Read entries from vector in internal representation.

Read entries of vector `mv` (stored in internal representation with modulus `p`) and store these entries in the array `b`. Here `len` is the number of rows to be read starting at row `i`. Each row consists of 32 entries.

Here `p` must be a legal modulus.

void **`mm_aux_write_mmv32`**(uint32_t p, uint8_t *b, *uint_mmv_t* *mv, uint32_t i, uint32_t len)

Write data to a vector in internal representation.

Write data from the array `b` to the vector `mv` (stored in internal representation with modulus `p`). Here `len` is the number of rows to be read starting at row `i`. Each row consists of 32 entries.

Here `p` must be a legal modulus.

void **mm_aux_read_mmv24**(uint32_t p, *uint_mmv_t* *mv, uint32_t i, uint8_t *b, uint32_t len)

Read rows of length 24 from vector in internal representation.

Read entries of vector `mv` with modulus `p` and store these entries in the array `b`, , starting at row `i`. Here `mv` is a vector of rows of 24 entries, with 8 entries slack after each row. `len` is the number of such rows to be read. So altogether 24 * `len` entries are read from `mv` and written to array `b`; the 8 bytes slack after each 24-byte row are dropped. Vector `b` is reduced modulo `p`.

Here `p` must be a legal modulus.

void **mm_aux_write_mmv24**(uint32_t p, uint8_t *b, *uint_mmv_t* *mv, uint32_t i, uint32_t len)

Write rows of length 24 to vector in internal representation.

Write data from the array `b` to the vector `mv` with modulus `p`. We take `24 * len` bytes from the array `b` and write them to the vector `mv`, starting at row `i`. Here `mv` is considered as a vector of rows of 24 entries, with 8 entries slack after each row; so `len` is the number of such 24-byte rows to be written. The entries in the slack after each row written to `mv` are set to zero.

Here `p` must be a legal modulus.

uint32_t **mm_aux_mmv_size**(uint32_t p)

Return the size of a vector in internal representation.

The function returns the number of integers of type `uint_mmv_t` required to store a vector of the representation ρ_p (in internal representation) with modulus `p`.

The function returns 0 if `p` is illegal modulus.

uint32_t **mm_aux_int_fields**(uint32_t p)

Return number of entries stored in integer of type `uint_mmv_t`

The function returns the number of entries of a vector of the representation ρ_p (in internal representation) that can be stored in an integer of type `uint_mmv_t`.

The function returns 0 if `p` is illegal modulus.

uint32_t **mm_aux_v24_ints**(uint32_t p)

Return number of integers of type `uint_mmv_t` to store 24 entries.

The function returns the number of integers of type `uint_mmv_t` required to store a part of 24 entries of a vector of the representation ρ_p (in internal representation) with modulus `p`. Such parts of 24 entries arise naturally in the construction of ρ_p.

The function returns 0 if `p` is illegal modulus.

void **mm_aux_zero_mmv**(uint32_t p, *uint_mmv_t* *mv)

Zero a vector in internal representation.

The function sets all entries of the vector `mv` with modulus `p` in internal representation to zero.

void **mm_aux_random_mmv**(uint32_t p, *uint_mmv_t* *mv, uint64_t *seed)

Randomize a vector in internal representation.

The function randomizes all entries of the vector `mv` with modulus `p` in internal representation uniformly using the internal random generator in file `gen_random.c`. Parameter `seed` must be a seed for a random generator as described in file `gen_random.c`.

int32_t **mm_aux_reduce_mmv**(uint32_t p, *uint_mmv_t* *mv)

Reduce a vector in internal representation.

The function reduces all entries of the vector `mv` with modulus `p` in internal representation to a standard form, so that equal vectors are represented by equal arrays of integers.

Note that a zero entry in such a vector can be represented either by the bit string `0...0` or by `1...1`. This functions sets all zero entries of the vector to `0...0`.

The function returns 0 if it detects no error. It may return the following error codes:

-1: Bad modulus `p`

-2: A one bit outside a valid bit field for an entry has been found

int32_t **`mm_aux_reduce_mmv_fields`**(uint32_t p, *uint_mmv_t* *mv, uint32_t nfields)

Auxiliary function of function `mm_aux_reduce_mmv`

The function performs the same operation as function `mm_aux_reduce_mmv`. But instead of all entries of the vector `mv`, it reduces the first `len` entries only.

int32_t **`mm_aux_check_mmv`**(uint32_t p, *uint_mmv_t* *mv)

Check a vector in internal representation for errors.

The function checks all entries of the vector `mv` with modulus `p` in internal representation for errors. It returns 0 if it detects no error. It may return the following error codes:

-1: Bad modulus `p`

-2: A one bit outside a valid bit field for an entry has been found

-3: A subfield of 24 entries has an illegal nonzero entry at index >= 24

-4: The vector has an illegal nonzero diagonal entry

-5: The symmetric part of the vector is not actually symmetric

As a side effect, `mv` is reduced with function `mm_aux_reduce_mmv`.

void **`mm_aux_small24_expand`**(uint8_t *b_src, uint8_t *b_dest)

Convert part of vector from external to internal representation.

Conversion between the internal and the external representation of a vector is straightforward, except for entries with tags `A, B, C`. The entries with these tags are stored in the first 852 entries of the external representation. In the internal representation the entries with these tags are spread over three symmetric 24 times 24 times matrices.

This function maps the 852 entries of the array `b_src` (corresponding to tags `A, B, C`) to the array `b_dest` of size 3 * 24 * 24 (corresponding to three symmetric 24 times 24 times matrices). Function `mm_aux_write_mmv24` can be used to write the data from the array `b_dest` to the initial segment of the internal representation of a vector.

void **`mm_aux_small24_compress`**(uint8_t *b_src, uint8_t *b_dest)

Convert part of vector from internal to external representation.

Conversion between the internal and the external representation of a vector is straightforward, except for entries with tags `A, B, C`. The entries with these tags are stored in the first 852 entries of the external representation. In the internal representation the entries with these tags are spread over three symmetric 24 times 24 times matrices.

This function maps the 3 * 24 * 24 entries of the array `b_src` (corresponding to three symmetric 24 times 24 times matrices) to the 852 entries of the array `b_dest` (corresponding to tags `A, B, C` in external representation).

This reverses the effect of function `mm_aux_small24_expand`. Function `mm_aux_read_mmv24` can be used to read the data from the initial segment of the internal representation of a vector to the array `b_src`, before calling this function.

void **`mm_aux_mmv_to_bytes`**(uint32_t p, *uint_mmv_t* *mv, uint8_t *b)

Convert vector from internal to external representation.

Read all entries of vector `mv` (stored in internal representation with modulus `p`) and store these entries in the array `b` in external representation.

Output vector `b` is reduced modulo `p`. It must have length 196884.

void **`mm_aux_bytes_to_mmv`**(uint32_t p, uint8_t *b, *uint_mmv_t* *mv)

Convert vector from external to internal representation.

Read all entries of the array `b (of length 196884, containing a vector in external representation) and store these entries the vector `mv`. Here `mv` is a vector stored in internal representation with modulus `p`.

Any entry `x` in the array `b` must satisfy `0 <= x <= p`. The vector `mv` is an array of `n` integers of type `uint_mmv_t` with `n = mm_aux_mmv_size(p)`.

int32_t **`mm_aux_mmv_to_sparse`**(uint32_t p, *uint_mmv_t* *mv, uint32_t *sp)

Convert vector from internal to sparse representation.

Read all entries of vector `mv` (stored in internal representation with modulus `p`) and store these entries in the array `sp` in sparse representation. Each entry in the array `sp` represents a nonzero entry of the vector. The function returns the length of the output array `sp` or an negative value in case of error. Negative return values are as in function `check_mmv_buffer`.

Output vector `sp` is reduced modulo `p`. The buffer for array `sp` must have length 196884. Input vector `mv` is checked with function `check_mmv_buffer`.

void **`mm_aux_mmv_extract_sparse`**(uint32_t p, *uint_mmv_t* *mv, uint32_t *sp, uint32_t length)

Extract entries from a vector in internal representation.

The function extracts certain entries from the vector `mv` depending on the vector `sp`. Here `mv` is a vector stored in internal representation with modulus `p`. Vector `sp` is a vector of length `length` in sparse representation.

The entries of vector `sp` are updated with the corresponding entries of `mv`. If `sp` has an entry with a certain label then the coordinate of that entry is set to the corresponding coordinate of vector `mv`. If several entries of `sp` have the same label then the same coordinate is taken from `mv` several times.

Bit 7,…,0 of any entry of `sp` should be either 0 or `p`. If that value is 0 then the coordinate is read to bits 7,…,0 of that entry. If that entry is `p` then the negative coordinate is read instead. Other values of these bits are strongly discouraged; but technically we XOR the corresponding coordinate of vector `mv` to these bits; and we then change a result `p` to zero. There is a special case where this detail is relevant.

uint32_t **`mm_aux_mmv_get_sparse`**(uint32_t p, *uint_mmv_t* *mv, uint32_t sp)

Extract one entry of a vector in internal representation.

The statement `uint32_t sp1 = mm_aux_mmv_get_sparse(p, mv, sp);` is equivalent to

```
uint32_t sp1 = sp; mm_aux_mmv_extract_sparse(p, mv, &sp1, 1);
```

void **`mm_aux_mmv_add_sparse`**(uint32_t p, uint32_t *sp, uint32_t length, *uint_mmv_t* *mv)

Add vector in sparse rep to vector in internal representation.

The function adds a vector `sp` in sparse representation to a vector `mv` in internal representation with modulus `p`. Vector `sp` has length `length`, and each value `x` in an entry of vector `sp` must satisfy `0 <= x <= p`. Different entries in `sp` with the same index are added up.

void **`mm_aux_mmv_set_sparse`**(uint32_t p, *uint_mmv_t* *mv, uint32_t *sp, uint32_t length)

Set certain entries of a vector in internal representation.

The function sets certain entries of the vector `mv` depending on the vector `sp`. Vector `mv` is given in internal representation with modulus `p`. Vector `sp` is given in sparse representation and has length `length`.

If `sp` has an entry with a certain label then the corresponding entry of `mv` is set to to the value coded in that entry of `sp`. Each of these values `x` must satisfy `0 <= x <= p`. Duplicate entries in `sp` with the same label and different values are illegal; in that case the value of `mv` is undefined.

int32_t **mm_aux_mmv_extract_sparse_signs**(uint32_t p, *uint_mmv_t* *mv, uint32_t *sp, uint32_t n)

Extract signs of a vector in internal representation.

The function extracts the signs of certain entries of the vector `mv` depending on the vector `sp`. Vector `mv` is given in internal representation with modulus `p`. Vector `sp` is given in sparse representation and has length `n`.

Entry `sp[i]` specifies a multiple `c[i] * u[i]` of a unit vector `u[i]`. Let `m[i]` be the coordinate of vector `mv` with respect to the unit vector `u[i]`. We put `s[i] = 0` if `m[i] = c[i]` and `s[i] = 1` if `m[i] = -c[i]`. In all other cases we assign a random value 0 or 1 to `s[i]`. Then the function returns the sum of the values `s[i] << i`, where `i` ranges from `0` to `n - 1`.

int32_t **mm_aux_mmv_extract_x_signs**(uint32_t p, *uint_mmv_t* *mv, uint64_t *elem, uint32_t *a, uint32_t n)

Extract some bits of a vector in internal representation.

The function extracts the least significant bits of certain entries of the vector `mv` depending on the vector `a`. Vector `mv` is given in internal representation of the Monster with modulus `p`.

Vector `a` is a an array of `n` elements of the group `Q_x0` of structure 2^{1+24}, with each element given in **Leech lattice encoding**. Here each element of `Q_x0` must correspond to a Leech lattice vector of type 2; otherwise the function fails.

The array `elem` represents an element of the group `G_x0` of structure $2^{1+24}.Co_1$, given in **G_x0 representation**. Internally, the function transforms (i.e. conjugates) all elements of `Q_x0` in the array `a` with the element `elem`, i.e. it calculates the element `a'[i] = elem^-1 * a[i] * elem` of `Q_x0`. Then it extracts the least significant bit `b[i]` of the entry of the vector `mv` with the coordinate labelled by `a'[i]`.

Note that negating `a'[i]` corresponds to negating the coordinate with label `a'[i]` in the vector `mv`. Hence when negating `a'[i]` we also have to flip the bit `b[i]`. The function fails in case `a'[i] = 0`.

The function returns the sum of the values `b[i] << i`, where `i` ranges from `0` to `n - 1`. The function also fails in case `i > 31`. It returns a negative value in case of failure.

Assume that a vector `mv' = mv * q * g` is given with a known vector `mv`, a known `g` in `G_x0`, and an unknown `q` in `Q_x0`. Then the main use case of this function is to find the element `q` (up to sign) without modifying `mv'`.

int32_t **mm_aux_mul_sparse**(uint32_t p, uint32_t *sp, uint32_t length, int64_t f, uint32_t p1, uint32_t *sp1)

Scalar multiplication and modular reduction in sparse representation.

The function multiplies a vector `sp` in sparse representation with a factor `f` and reduces the result modulo a number `p1`.

The vector `sp` has length `length` and is stored in sparse representation as a vector modulo an odd number `2 < p < 256`. The result is reduced modulo the number `p1` and stored in sparse representation in the array `sp1`.

The number `p1` must be odd and satisfy `2 < p < 256`. In case `f != 0` the number `p1` must divide `p * abs(f)`.

The function returns the length of the array `sp1` in case of success and `-1` in case of failure.

The two arrays `sp` and `sp1` may be non overlapping or equal.

int32_t **mm_aux_get_mmv_leech2**(uint32_t p, *uint_mmv_t* *mv, uint32_t v2)

Read entry from vector in internal rep indexed by Leech lattice.

The function returns the entry with index `i` of the vector `mv` with modulus `p`. Here `i` must be an index referring to a vector of type 2 in the Leech lattice modulo 2 in **Leech lattice encoding**. The sign bit 24 of `i` is evaluated as expected. The return value is reduced modulo `p`.

The function returns a negative value if `i` is not a vector of type 2 in the Leech lattice modulo 2 or `p` is not a legal modulus.

uint64_t **mm_aux_hash**(uint32_t p, *uint_mmv_t* *mv)

Compute hash value of vector in internal.

The function returns a hash value of the vector `mv` with modulus `p`. It also tries to distinguish between different sparse vectors. Therefore it tries to hash over about 100 nonzero integers of type `uint_mmv_t`. So if `mv` is sparse then the function might have to scan considerably more zero entries.

3.5.6 C interface for file mm_tables.c

File `mm_tables.c` contains functions for supporting the operations $x_\epsilon x_\pi$ and $y_f x_e x_\epsilon$ on the vectors of the 198884-dimesional represention ρ_p modulo a small number p. These operations are implemented in separate packages for the differnet values of p. But there are commom preprocessing steps required for all values p. These common preprocessing steps are implemented here.

The monomial operations $x_\epsilon, x_\pi, y_f, x_e$ are as defined in the **API Reference**, section **The monster group**. Here ϵ is an element of the Golay cocode represented as a 12-bit integer. f and e are elements of the Parker loop represented as 13-bit integers. π is a automorphism of the Parker loop. In the **API Reference**, section **Automorphisms of the Parker loop** we number a certain set of these automorphisms in the same way as the elements of the Mathieu group M_{24}. We denote such an automorphism π by its number.

Functions

void **mm_sub_prep_pi**(uint32_t eps, uint32_t pi, *mm_sub_op_pi_type* *p_op)

Compute information for operation $x_\epsilon x_\pi$.

Given an element $x_\epsilon x_\pi$ of the monster by parameters `eps` and `pi`, the function computes the relevant data for performing that operation on the representation of the monster. These data are stored in the structure of type *`mm_sub_op_pi_type`* referred by parameter `p_op`.

Caution:

Component `p_op->tbl_perm64` must be initialized with an array of type *`mm_sub_op_pi64_type`* `a_tbl_perm64[759]` befor aclling this function!!!.

int32_t **mm_sub_test_prep_pi_64**(uint32_t eps, uint32_t pi, uint32_t *p_tbl)

For internal use only!

This is an auxiliary function for testing function `mm_sub_prep_pi`.

Given `eps` and `pi` as in function `mm_sub_prep_pi`, this function executes function `mm_sub_prep_p(eps, pi, p_op)` and stores the output `p_op->tbl_perm64` (as an array of length 759 * (1 + 6)) in the array referred by `p_tbl`.

void **mm_sub_prep_xy**(uint32_t f, uint32_t e, uint32_t eps, *mm_sub_op_xy_type* *p_op)

Compute information for operation $y_f x_e x_\epsilon$.

Given an element $y_f x_e x_\epsilon$ of the monster by parameters `f`, `e`, and `eps`, the function computes the relevant data for performing that operation on the representation of the monster. These data are stored in the structure of type *`mm_sub_op_xy_type`* referred by parameter `p_op`.

Caution!

Component `p_op->sign_XYZ` must either be NULL or refer to an array of type `uint8_t` of length 2048. Component `p_op->s_T` must either be NULL or refer to an array of type `uint8_t` of length 759.

3.5.7 C interface for file mm_op_p_vector.c

File `mm_op_p_vector.c` implements the operation of the Monster group on a vector in the representation ρ_p of the Monster modulo p.

The representation ρ_p is equal to the 196884-dimensional representation ρ of the monster, with coefficients taken modulo p, as defined in section **The representation of the monster group** in the **API reference**.

Unless otherwise stated, the first parameter of a function in this module is the modulus p, and that modulus must be one of the values 3, 7, 15, 31, 127, or 255.

An element of ρ_p is implemented as an array of integers of type `uint_mmv_t` as described in section **Description of the mmgroup.mm extension** in this document.

The number of entries of a vector of type `uint_mmv_t[]` for modulus p is equal to `mm_aux_mmv_size(p)`.

Functions

int32_t **`mm_op_copy`**(uint32_t p, *uint_mmv_t* *mv1, *uint_mmv_t* *mv2)

Copy vector `mv1` in ρ_p to `mv2`

Here modulus `p` must be one of the values 3, 7, 15, 31, 127, or 255.

int32_t **`mm_op_compare_len`**(uint32_t p, *uint_mmv_t* *mv1, *uint_mmv_t* *mv2, uint32_t len)

Compare arrays `mv1` and `mv2` of integers.

The function compares parts of the two vectors `mv1` and `mv2`of the representation ρ_p.

Here the function compares `len` integers of type `uint_mmv_t` starting at the pointers `mv1` and `mv2`. These integers are interpreted as arrays of bit fields containing integers modulo p.

The function returns 0 in case of equality and 1 otherwise.

Here modulus `p` must be one of the values 3, 7, 15, 31, 127, or 255.

int32_t **`mm_op_compare`**(uint32_t p, *uint_mmv_t* *mv1, *uint_mmv_t* *mv2)

Compare vectors `mv1` and `mv2` of ρ_p.

The function compares two vectors `mv1` and `mv2` of the representation ρ_p.

It returns 0 in case of equality and 1 otherwise.

Here modulus `p` must be one of the values 3, 7, 15, 31, 127, or 255.

int32_t **`mm_op_checkzero`**(uint32_t p, *uint_mmv_t* *mv)

Check if a vector `mv` in ρ_p is zero.

The function checks it the vector `mv` in the representation ρ_p is zero.

It returns 0 in case `mv == 0` and 1 otherwise. It is optimized for the case that `mv` is expected to be zero.

Here modulus `p` must be one of the values 3, 7, 15, 31, 127, or 255.

int32_t **`mm_op_vector_add`**(uint32_t p, *uint_mmv_t* *mv1, *uint_mmv_t* *mv2)

Add vectors `mv1` and `mv2` of ρ_p.

The function adds the two vectors `mv1` and `mv2` of the representation ρ_p and stores the result in the vector `mv1`.

Here modulus `p` must be one of the values 3, 7, 15, 31, 127, or 255.

int32_t **`mm_op_scalar_mul`**(uint32_t p, int32_t factor, *uint_mmv_t* *mv1)

Multiply vector `mv1` of ρ_p with scalar.

The function multiplies the vector `mv1` of the representation ρ_p and with the (signed) integer `factor` and stores the result in the vector `mv1`.

Here modulus `p` must be one of the values 3, 7, 15, 31, 127, or 255.

int32_t **`mm_op_compare_mod_q`**(uint32_t p, *uint_mmv_t* *mv1, *uint_mmv_t* *mv2, uint32_t q)

Compare two vectors of ρ_p modulo q.

The function compares two vectors `mv1` and `mv2` of the representation ρ_p modulo a number q. Here q should divide p.

It returns 0 in case of equality, 1 in case of inequality, and 2 if q does not divide p.

Here modulus `p` must be one of the values 3, 7, 15, 31, 127, or 255.

int32_t **`mm_op_store_axis`**(uint32_t p, uint32_t x, *uint_mmv_t* *mv)

Set a vector in ρ_p to an axis.

Let `x` be an element of the subgroup Q_{x0} if the Monster that maps to a short Leech lattice vector. Here `x` must be given in **Leech lattice encoding** as in the **Description of the mmgroup.generators extension** in the documentation of the **C interface**.

Then `x` corresponds to vector in ρ_p that is called a **2A axis**. The function stores that 2A axis in `mv`.

Here modulus `p` must be one of the values 3, 7, 15, 31, 127, or 255.

int32_t **`mm_op_pi`**(uint32_t p, *uint_mmv_t* *v_in, uint32_t delta, uint32_t pi, *uint_mmv_t* *v_out)

Compute automophism of the Parker loop on a vector.

File `mm_op_pi.c` implements the operation of the generators x_π and x_δ of the monster group on a vector in the representation ρ_p of the Monster modulo `p`. Here `p` must be 3, 7, 15, 31, 127, or 255.

Here generators x_π and x_δ are defined as automorphisms of the Parker loop as in section **The monster group** of the **API reference**. An automophism of the Parker loop is specified by a pair of integers `d, pi` as in the constructor of the Python class `AutPL`, see section **Automophisms of the Parker loop** in the **API reference**.

The exact operation of an automorphism of the Parker loop on ρ is as defined in [Seysen19].

Note that the integers `d, pi` mentioned above describe the number of an element of the Golay cocode and the number of a permutation in the Mathieu group M_{24}, respectively. Internally, we use the C functions in file `mat24_functions.c` and the function `mm_sub_prep_pi` in file `mm_tables.c` for converting the integers `d, pi` to mathematical objects that can be used for implementing the operation on ρ_p. These conversions are very fast compared to the cost for the operation on ρ_p. This helps us to keep the C interface for these operations simple.

Let `v_in` be a vector of the representation ρ_p of the monster group. Then the function computes this automorphism on the input vector `v_in` and stores the result in the output vector `v_out.` Input vector `v_in` is not changed.

Here modulus `p` must be one of the values 3, 7, 15, 31, 127, or 255.

int32_t **`mm_op_xy`**(uint32_t p, *uint_mmv_t* *v_in, uint32_t f, uint32_t e, uint32_t eps, *uint_mmv_t* *v_out)

Compute an operation of the monster group on a vector.

Let `v_in` be a vector of the representation ρ_p of the monster group.

The function implements the operation of the element $y_f \cdot x_e \cdot x_\epsilon$ of the monster group on a vector `v_in` in the representation ρ_p of the monster.

The integers `f` and `e` occuring in the generators y_f and x_e encode elements of the Parker loop. The integer `eps` encodes the element ϵ of the Golay cocode occuring in the generator x_ϵ, as indicated in the header of this file. The function computes this operation of the element of the monster (given by parameters `f, e, eps`) on the input vector `v_in` and stores the result in the output vector `v_out.`

Input vector `v_in` is not changed.

Here modulus `p` must be one of the values 3, 7, 15, 31, 127, or 255.

int32_t **mm_op_omega**(uint32_t p, *uint_mmv_t* *v, uint32_t d)

Compute an operation of the monster group on a vector.

Let `v` be a vector of the representation ρ_p of the monster group.

The function implements the operation of the element x_d of the monster group on a vector `v` in the representation ρ_p of the monster. Here `d` must be one of the integers `0, 0x800, 0x1000`, or `0x1800`, encoding the generators x_1, x_Ω, x_{-1}, or $x_{-\Omega}$, respectively.

The function computes the operation x_d on the vector `v` and overwrites the vector `v` with the result. The function can be considered as a simplified (and much faster) version of function `mm_op_xy`.

Here modulus `p` must be one of the values 3, 7, 15, 31, 127, or 255.

int32_t **mm_op_t_A**(uint32_t p, *uint_mmv_t* *v_in, uint32_t e, *uint_mmv_t* *v_out)

Compute part A of operation of τ^e on vector``.

Function `mm_op_t_A` computes a the operation of the monster group element τ^e on a vector `v_in` and stores the A part of the result in a vector `v_out`. That operation depends on a parameter `e`. The other entries of vector `v_out` are not changed. See section **The representation of the monster group** in the **API reference** for tags of entries of a vector in the representation of the monster. Note that the entries of vector `v_out` with tag `A` also depend on entries of vector `v_in` with tags different from `A`.

Here modulus `p` must be one of the values 3, 7, 15, 31, 127, or 255.

int32_t **mm_op_word**(uint32_t p, *uint_mmv_t* *v, uint32_t *g, int32_t len_g, int32_t e, *uint_mmv_t* *work)

Compute operation of the monster group on a vector.

Let v be a vector of the representation ρ_p of the monster group stored in the array referred by `v`.

Let g be the element of the monster group stored in the array of length `len_g` referred by the pointer `g`.

Then the function computes the vector $v \cdot g^e$ and overwrites the vector in the array `v` with that vector. Here e is the exponent given by the integer `e`.

The function requires a work buffer (referrd by `work`), which is an array of `mm_aux_mmv_size(p)` entries of type `uint_mmv_t`. So the work buffer has the same size as the vector `v`.

The function returns 0 in case of success and a nonzero value in case of failure.

Internally, the function simplifies all substrings of the string representing the word g^e, except for atoms corresponding to nonzero powers of the generator ξ. So the user need not 'optimize' the input g. Of course, this simplification does not change the input array `g`.

Here modulus `p` must be one of the values 3, 7, 15, 31, 127, or 255.

int32_t **mm_op_word_tag_A**(uint32_t p, *uint_mmv_t* *v, uint32_t *g, int32_t len_g, int32_t e)

Restriction of function `mm_op_word` to tag `A`

Function `mm_op_word` computes the operation of an element $h = g^e$ of the monster group a vector `v` and overwrites `v` with the result of that operation. h depends on parameters `g, len_g,` and `e` of this function.

Function `mm_op_word_tag_A` computes the same automorphism on the entries of the vector `v` with tag `A` only, and ignores the other entries of `v`. See section **The representation of the monster group** in the **API reference** for tags of entries of a vector in the representation of the monster.

The function overwrites the vector `v` with the result. Here only entries of `v` with tag `A` are changed.

Parameters and return value are the same as in function `mm_op_word`, except that a work buffer is not required here. Also, the function fails and returns a nonzero value, if the word that makes up the group element h contains any nonzero powers of the generator τ of the monster group. Note that such a power of τ does not fix the the part of the vector `v` with tag `A`. Powers of the generator τ correspond to atoms with tag `t`.

This function is much faster than function `mm_op_word`. Array `v` must have `24 * mm_aux_v24_ints(p)` entries of type `uint_mmv_t`.

Here modulus `p` must be one of the values 3, 7, 15, 31, 127, or 255.

int32_t **`mm_op_word_ABC`**(uint32_t p, *uint_mmv_t* *v, uint32_t *g, int32_t len_g, *uint_mmv_t* *v_out)

Compute ABC part of the operation of the monster group on a vector.

Let v be a vector of the representation ρ_p of the monster group stored in the array referred by `v`.

Let g be the element of the monster group stored in the array of length `len_g` referred by the pointer `g`. Here g, and also all prefixes of the word representing g, must be in the set $G_{x0} \cdot N_0$.

The function computes the parts with tags `A`, `B`, and `C` of the vector $v \cdot g$ and stores the result in the array `v_out`. The other parts of the vector $v \cdot g$ are not computed. Here the array `v_out` must have `72 * mm_aux_v24_ints(p)` entries of type `uint_mmv_t`.

This function is much faster than function `mm_op_word`. It is mainly used for dealing with a 2A axis v.

Here modulus `p` must be one of the values 3, 7, 15, 31, 127, or 255.

int32_t **`mm_op_scalprod`**(uint32_t p, *uint_mmv_t* *v1, *uint_mmv_t* *v2)

Compute the scalar product of two vectors.

Let v_1, v_2 be vectors of the representation ρ_p of the monster group stored in the array referred by `v1, v2`.

The function returns the scalar product (v_1, v_2) (reduced modulo p) in case of success and a negative value in case of failure.

Here modulus `p` must be one of the values 3, 7, 15, 31, 127, or 255.

Variables

const uint8_t **`MM_OP_P_TABLE`**[] = {0x03, 0x07, 0x0f, 0x1f, 0x7f, 0xff, 0x00}

Table of legal moduli `p`, terminted by zero.

3.5.8 C interface for file mm_op_p_axis.c

File `mm_op_p_axis.c` implements the operation of the Monster group on a vector in the representation ρ_p of the Monster modulo p. In this module such a vector is usually a 2A axis.

The representation ρ_p is equal to the 196884-dimensional representation ρ of the monster, with coefficients taken modulo p, as defined in section **The representation of the monster group** in the **API reference**.

The first parameter of a function in this module is the modulus p, and that modulus must be one of the values specified in the function.

An element of ρ_p is implemented as an array of integers of type `uint_mmv_t` as described in section **Description of the mmgroup.mm extension** in this document.

The number of entries of a vector of type `uint_mmv_t[]` for modulus p is equal to `mm_aux_mmv_size(p)`.

Functions

int32_t **mm_op_load_leech3matrix**(uint32_t p, *uint_mmv_t* *v, uint64_t *a)

Load the 'A' part of a vector of the representation of the monster.

The function loads the part of with tag 'A' of a vector `v` of the representation of the monster modulo `p` to the matrix `a`. Here matrix `a` will be given in **matrix mod 3** encoding as documented in the header of file `leech3matrix.c`.

Here modulus `p` must be one of the values 3, or 15.

int64_t **mm_op_eval_A_rank_mod3**(uint32_t p, *uint_mmv_t* *v, uint32_t d)

Rank of 'A' part of a vector of the representation of the monster.

Let `a` be the symmetric 24 times matrix corresponding to the part with tag 'A' of a input vector `v` in the representation of the monster modulo `p`. Let `b = a - d * 1`, for an integer input `d`, where `1` is the unit matrix.

Let `r` be the rank of matrix `b` with entries taken modulo 3. If matrix `b` has rank 23 then its kernel is one dimensional. In that case the kernel contains two nonzero vectors `+-w`, and we define `w` to be one of these vectors. Otherwise we let `w` be the zero vector.

The function returns the value `(r << 48) + w`, with `w` the vector defined above given in *Leech lattice mod 3 encoding* as described in *The C interface of the mmgroup project*.

The number of entries of parameter `v` for modulus p is equal to `24 * mm_aux_int_fields(p)`.

Here modulus `p` must be 3, or 15.

int32_t **mm_op_eval_A_aux**(uint32_t p, *uint_mmv_t* *v, uint32_t m_and, uint32_t m_xor, uint32_t row)

Auxiliary function for `mm_op_eval_A`

Let matrix `A` be the part with tag 'A' of a vector `v` of the representation of the monster modulo `p}`.

Let `m_and[i]` and `m_xor[i]` be the bit `i` of `m_and` and `m_xor`, respectively. Define a vector `y = (y[0],... ,y[23])` by: `y[i] = m_and[i] * (-1)**m_xor[i]`.

If `row >= 24` the function returns `res = y * A * transpose(y)` (modulo `p`). We have `0 < res < 0x8000`, but `res` is not reduced modulo `p`.

In case `row < 24` define the vector `z` by `z[i] = y[i]` if `i = row` and `z[i] = 0` otherwise. Put `zz = z * A * transpose(y)` (modulo `p`). We have `0 < res < 0x8000`, but `res` is not reduced modulo `p`.

Here modulus `p` must be 3, or 15.

int32_t **mm_op_eval_A**(uint32_t p, *uint_mmv_t* *v, uint32_t v2)

Evaluate A part in rep of monster at a short Leech vector.

Let `v` be a vector in the 196884-dimensional representation of the monster group modulo `p`, encoded as described in section *Description of the mmgroup.mm<p> extensions* in the description of the *C interface*. The entries corresponding to tag 'A' of `v` form a symmetric 24 times 24 matrix A.

Let v_2 be a short Leech lattice vector given by parameter `v2`, encoded as a vector in the Leech lattice modulo 2. Then v_2 is determined up to sign and $v_2 A v_2^\top$ is determined uniquely.

The function returns $r = v_2 A v_2^\top$ modulo `p`, with $0 \leq r < p$ in case of success. It returns -1 if v_2 is not short (i.e. not of type 2).

The short Leech lattice vector v_2 (of norm 4) is scaled to norm 32 as usual, when v_2 is given in integer coordinates.

Here modulus `p` must be 3, or 15.

int32_t **mm_op_norm_A**(uint32_t p, *uint_mmv_t* *v)

Compute norm of the 'A' part of a vector in the rep of the monster.

Assume that v is a vector in the representation of the monster modulo p. Then the part of v with tag 'A' is considered as a symmetric 24 times 24 matrix. The function returns the norm (i.e. the sum of the squares of the entries) of that matrix.

Here modulus p must be 3, or 15.

int32_t **mm_op_watermark_A**(uint32_t p, *uint_mmv_t* *v, uint32_t *w)

Watermark 'A' part of a vector of the representation of the monster.

Let matrix A be the part with tag 'A' of a vector v of the representation of the monster modulo p.

Then we watermark 24 the rows of matrix A. For each of the rows A[i], 0 <= i < 24 we compute a watermark w(i) in the array w. Note that the watermark w(i) contains an information about the marked row i in its lower bits. We store the sorted array of these watermarks in the array w of length

a. If all these watermarks (ignoring the information about the row) are different, we can easily recognize a permutation of the rows of matrix A by comparing the watermark of matrix A with the watermark of the permuted matrix A.

The watermark w[i] depends on the distribution of the absolute values of the entries w[i, j] (modulo p) of row i. Thus permutations of the columns and sign changes in the matrix do not affect these watermarks.

The function returns 0 in case of success and a negative value in case of error.

When working in the representation modulo p = 3 we fail unless at least nine rows of A have a unique watermark. This is sufficient for reconstructing a permutation in the Mathieu group.

In the other cases the watermark of row i is equal to $i+32 \cdot S(A, i)$. Here $S(A, i)$ depends on the entries of matrix A. The value $S(A, i)$ it is invariant under sign changes of any off-diagonal elements of A. It is also invariant under any permutation of the symmetric matrix A fixing row and column i.

We assert that watermarking A + k*I succeeds if and only if watermarking A succeeds, for any multiple k*I of the unit matrix.

Here modulus p must be 3, or 15.

int32_t **mm_op_watermark_A_perm_num**(uint32_t p, uint32_t *w, *uint_mmv_t* *v)

Compute permutation from watermarks of matrices.

Let matrix A be the part with tag 'A' of a vector v of the representation of the monster modulo p. Let w be the watermark of another matrix A' which is obtained from A by permutations of the rows and columns, and by sign changes. Here the watermark w must have been computed by function mm_op_watermark_A.

Then the function watermarks matrix A and computes a permutation that maps A' to A. If that permutation is in the Mathieu group M_{24} then the function returns the number of that permutation, as given by function mat24_perm_to_m24num in file mat24_functions.c.

The function returns a nonegative permutation number in case of success and a negative value in case of error.

If all watermarks in the array w (ignoring the information about the row in the lower 5 bits) are different then there is at most one permutation that maps A' to A. If that permutation is in M_{24} then the function returns the number of that permutation. In all other cases the function fails.

In case p = 3 we succeed already if the first 9 watermarks in the array w are different. Then there is at most one permutation in M_{24} that maps A' to A.

Here modulus p must be 3, or 15.

int32_t **mm_op_eval_X_find_abs**(uint32_t p, *uint_mmv_t* *v, uint32_t *p_out, uint32_t n, uint32_t y0, uint32_t y1)

Find certain entries of a vector of the monster rep modulo `%p`

Let `v` be a vector of the monster group representation modulo `%p`. The function tries to find all entries of the monomial part of `v` with absolute values `y0` and `y1`, `1 <= y0, y1 <= p / 2`. In case `y1 = 0` entries with value `y1` are ignored.

Here the monomial part of `v` consists of the entries with tags 'B', 'C', 'T', 'X'. The coordinates of these entries correspond to the short vectors of the Leech lattice.

Output is written into the array `p_out` of length `n`. If the monomial part of `v` contains an entry with absolute value `y0` then the coordinate of that entry is written into array `p_out` in **Leech lattice encoding**. If that part of `v` contains an entry with absolute value `y1` then the coordinate of that entry is witten into that array in the same encoding.

In addition, for entries in `v` with absolute value `y1` the bit 24 of the corresponding entry in `p_out` is set. In `p_out`, the entries with absolute value `y1` are stored after those with absolute value `y0`. Entries with the same absolute value are stored in the same order as in `v`.

The function returns the number of valid entries in the array `p_out`. If the length `n` of `p_out` is too small then some entries will be dropped without notice.

Here modulus `p` must be 15.

int32_t **mm_op_eval_X_count_abs**(uint32_t p, *uint_mmv_t* *v, uint32_t *p_out)

Count certain entries of a vector of the monster rep modulo `p`

Let `v` be a vector of the monster group representation modulo `p`. The function counts the absolute values of all entries of the monomial part of `v`.

Here the monomial part of `v` consists of the entries with tags 'B', 'C', 'T', 'X'. The coordinates of these entries correspond to the short vectors of the Leech lattice.

Output is written into the array `p_out` of length `p/2 + 1`. Entry `p_out[i]` contains the number if entries of the monomial part of `v` with absolute value `i` for `0 <= i <= p/2 + 1`.

Here modulus `p` must be 15.

The function returns 0.

3.5.9 Internal operation

The source code for the functions in modules `mm_aux.c` and `mm_tables.c` is located in subdirectory `src/mmgroup/dev/mm_basics`. The code generation process genrates `.c` files from the source files with extension `.ske` in that subdirectory. Protoypes for the functions in these `.c` files can be found in file `mm_basics.h`.

The source code for the functions in modules `mm_op_p_vector.c` and `mm_op_p_axis.c` is located in subdirectory `src/mmgroup/dev/mm_op`. The code generation process genrates `.c` files from the source files with extension `.ske` in that subdirectory. Protoypes for the functions in these files can be found in file `mm_o_p.h`.

The code gerneration process for C files derived from the source files in the `src/mmgroup/dev/mm_basics` directory is straightforward. In the remainder of this subsection we describe the code gerneration process for files derived form sources in the `src/mmgroup/dev/mm_op` directory.

For reasons of efficiency a dedicated set of `.c` files is generated from the `.ske```files in the ``src/mmgroup/dev/mm_op` directory for each modulus `p` supported. Here corresponding `.c` files for different moduli are generated from the same source file with extension `.ske`. The `.c` functions in modulea `mm_op_p_vector.c` and `mm_op_p_axis.c``contain automatically-generated ``case` statements that call the working functions for the appropriate modulus `p`. Any of these `.c` functions takes the modulus `p` as its first argument, and dispatches the work to the subfunction dedicated to the appropriate modulus `p`.

E.g. the function `mm_op_pi` in module `mm_op_p_vector.c` calls working functions ``mm3_op_pi``, `mm7_op_pi`, etc. in case `p = 3`, `p = 7`, etc., for doing the actual work in the representation of the Monster modulo `p`. Such working functions are implemented in different `.c` files, with one `.c` file for each modulus. So the functions `mm3_op_pi` and `mm7_op_pi` are implemented in the `.c` files `mm3_op_pi.c` and *mm7_op_pi.c*`, respectively. Here such a `.c` file may implement several working functions for the same modules `p`.

All functions exported from file `mm_op_p_vector.c` support the same set of moduli `p`. Functions exported from file `mm_op_p_axis.c` deal with 2A axes; the usually support moduli 3 and 15 (or just one of these two values) only; see documentation of the corresponding functions.

So the process of switching to the appropriate function for a given modulus is completely invisible for Python and Cython.

3.5.10 Basic table-providing classes for module `mmgroup.mm_op`

We describe the class `MM_Basics` and its subclass `MM_Const`

Class `MM_Basics` in module `mmgroup.dev.mm_basics.mm_basics` defines several constants that describe the organization of vectors of integers modulo p used in the representation modulo ρ_p. Here several integers modulo p are stored in a single unsigned integer of type `uint_mmv_t`. Type `uint_mmv_t` is equal to the standard C integer type `uint64_t` on a 64-bit system.

For a 32-bit system that type may be changed to `uint32_t` by adjusting the variable `INT_BITS` in this module, but this has no longer been tested for several years.

Table 4: Constants used for generating representations of the monster

Name	Value
`FIELD_BITS`	Number of bits used to store an integer modulo `P`. This is the smallest power of two greater than or equal to `P_BITS`.
`INT_BITS`	Number of bits available in an unsigned integer of type `uint_mmv_t`. This is equal to 64.
`INT_FIELDS`	Number of integers modulo `P` that is stored in a variable of type `uint_mmv_t`. `INT_FIELDS` is a power of two and equal to `INT_BITS/FIELD_BITS`.
`LOG_FIELD_BITS`	Binary logarithm of the value `FIELD_BITS`.
`LOG_INT_BITS`	Binary logarithm of the value `INT_BITS`.
`LOG_INT_FIELDS`	Binary logarithm of the value `INT_FIELDS`.
`LOG_V24_INTS`	Binary logarithm of the value `V24_INTS`.
`LOG_V64_INTS`	Binary logarithm of the value `V64_INTS`.
`MVV_ENTRIES`	Number of integers modulo `P` contained in a vector of the representation of the monster, including unused entries.
`MVV_INTS`	Number of integers of type `uint_mmv_t` required to store a vector of the representation of the monster. This value depends on `P`.
`P`	The modulus of the current representation being generated. `P` is equal to `2**P_BITS - 1`.
`P_BITS`	Bit length of modulus `P`.
`V24_INTS`	Number of integers of type `uint_mmv_t` used up to store a vector of `24` integers modulo `P`. `V24_INTS` is always a power of two.
`V64_INTS`	Number of integers of type `uint_mmv_t` used to store a vector of `64` integers modulo `P`. `V64_INTS` is always a power of two.
`V64_INTS_USED`	Number of integers of type `uint_mmv_t` actually used to store a vector of `24` integers modulo `P`. This may be less than `V24_INTS`.

Class `MM_Const` in module `mmgroup.dev.mm_basics.mm_basics` is a table-providing class for the C functions used by python extension `mmgroup.mm`. Most C functions in that module take the modulus `p` as a parameter. They need fast access to the constants given in the table above for all legal values of `p`. Using the code generator with the tables and directives provided by class `MM_Const`, we can easily obtain these values for any legal `p` as in the following example:

```
// Return a nonzero value if p is a bad modulus,
// i.e. not p = 2**k - 1 for some 2 <= k <= 8
#define mm_aux_bad_p(p) (((p) & ((p)+1)) | (((p)-3) & ((0UL-256UL))))

// Create a precomputed table containing the constants for modulus p
// %%USE_TABLE
static const uint32_t MMV_CONST_TABLE[] = {
// %%TABLE MMV_CONST_TAB, uint32
};

int do_something_for_modulus_p(uint 32_t p, ...)
{
    uint32_t modulus_data, p_bits, field_bits;
    // Reject any illegal modulus p
    if (mm_aux_bad_p(p)) return -1;
    // Load the constants for modulus p to variable modulus_data
    // %%MMV_LOAD_CONST  p, modulus_data;
    // Load value P_BITS for modulus P to p_bits (using modulus_data)
    p_bits = %{MMV_CONST:P_BITS,modulus_data};
    // Load value FIELD_BITS for modulus P to field_bits
    field_bits = %{MMV_CONST:FIELD_BITS,modulus_data};
    // Now do the actual work of the function using these values
    ...
}
```

class `mmgroup.dev.mm_basics.mm_basics.`**`MM_Const`**(***kwds*)

This is the basic table-providing class for module `mmgroup.mm`

The main purpose of this class is to provide the constants defined in class `MM_Basics, for a variable modulus ``p` as shown in the example above.

This class provides the directive `MMV_CONST_TAB` for generating all tables, and the directive `MMV_LOAD_CONST` for storing the table of constants, for a specific modulus `p`, in an integer variable. The string formatting function `MMV_CONST` can be used for extracting a specific constant from that variable, as indicated in the example above.

Internally, we use a deBruijn sequence to translate the value `p` to an index for the table generated via the directive `MMV_CONST_TAB`.

Constants not depending on the modulus `p`, such as `INT_BITS`, `LOG_INT_BITS`, and `MMV_ENTRIES` are available as attributes of class `MM_Const`. They can also be coded with the code generator directly via string formatting, e.g.:

```
uint_8_t  a[%{MMV_ENTRIES}];
```

For a fixed modulus `p` the constants depending on `p` can also be coded with the code generator via string formatting, e.g.:

```
a >>= %{P_BITS:3};
```

That constant also availble in the form `MM_Const().P_BITS(3)`.

Class `MM_Const` provides a string-formatting function `shl` which generates a shift expression Here:

```
%{shl:expression, i}
```

generates an expression equivalent to `((expression) << i)`. Here `i` must be an integer. In case `i < 0` we generate `((expression) >> -i)` instead. E.g. `%{shl:'x',-3}` evaluates to `x >> 3`.

Class `MM_Const` provides another string-formatting function `smask` which generates an integer constant to be used as a bit mask for integers of type `uint_mmv_t`. Here:

```
%{smask:value, fields, width}
```

with integers `value`, `fields`, and `width` creates such a bit mask. `For 0 <= i` and `0 <= j < width`, the bit of that mask at position `i * width + j` is set if both, bit `i` of the integer `fields` and bit `j` of the integer `value` are set. A bit in the mask at position `INT_BITS` or higher is never set.

Any of the arguments `value` and `fields` may either be an integer or anything iterable that yields a list of integers. Then this list is interpreted as a list of bit positions. A bit of that argument is set if its position occurs in that list and cleared otherwise.

E.g. `%{smask:3, [1,2], 4}` evaluates to `0x330`.

`snippet`(*source*, **args*, ***kwds*)

Generate a C code snippet

This method calls function `c_snippet` in module `mmgroup.generate_c` to generate a C code snippet. Parameters are as in function `c_snippet`. The method returns the generated C code snippet as a string.

In addition, all tables and directives available in the given instance of the table-providing class are also passed as arguments to function `c_snippet`.

We describe the class `MM_Op`.

Class `MM_Op` in module `mmgroup.dev.mm_op.mm_op` is a table-providing class for the C functions used by python extensions `mmgroup.mm<p>`. Each of these extensions implements the operation of monster group modulo a fixed number `p`.

Class `MM_Op` provides the same functionality as class `MM_Const` in module `mmgroup.dev.mm_basics.mm_basics`. Since modulus `p` is fixed in an instance of class `MM_Op`, all constants provided by the base class `MM_Basics` of `MM_Op` and `MM_Const` (see table *Constants used for generating representations of the monster*) have fixed values. So they are available as attributes of an instance of class `MM_Op` and they may also be used by the code generator directly via string formatting.

The string formatting functions `%{shl:expression,i}` and `%{smask:value, fields, width}` work as in class `MM_Op`. In addition, parameter `fields` defaults to `-1` (i.e. the mask is set in all bit fields) and `width` defaults to the constant `FIELD_BITS`. i.e. the number of bits used to store an integer modulo `p`. E.g. for `p = 7` we have `FIELD_BITS = 4`, and `%{smask:1}` evaluates to the 64-bit integer constant `0x1111111111111111`.

Class `MM_Op` also provides the following directives:

`MMV_ROTL src, count, dest`

Rotate the bit fields of a variable of type `uint_mmv_t`

Here `src` is an integer of type `uint_mmv_t` which is interpreted as an array of bit fields, where each bit field stores a number modulo `p`. Then each bit field is rotated left by `count` bits. This means that the numbers in all bit fields are multiplied by `2**count`. `count` must be an integer. It is reduced modulo the bit length of `p`. So `count` may also be negative. The result of the rotation is written to the variable `dest` which should also be of type `uint_mmv_t`. `dest` defaults to `src`.

`MMV_UINT_SPREAD src, dest, value`

Spread bits of an integer `src` to the bit fields of `dest`. Here `src` may be any integer variable and `dest` should be a variable of type `uint_mmv_t`. If bit `i` of the integer `src` is set then the `i`-th bit field of variable `dest` is set to `value`; otherwise it is set to zero. `value` must be an integer with `0 <= value <= p`; default is `p`.

class `mmgroup.dev.mm_op.mm_op.`**`MM_Op`**(***kwds*)

Supports basic operations on integers of type `uint_mmv_t`.

Here an integer of type `uint_mmv_t` is interpreted as an array of bit fields, where each bit field stores an integer modulo `p` for a fixed number `p`. This class is similar to class `MM_Const` in module `mmgroup.dev.mm_basics.mm_basics`, where the modulus `p` may be variable.

Usage of this class is documented in the module documentation.

Parameters

p (*int*) – This is the modulus `3 <= p < 256`. `p + 1` must be a power of two.

3.5.11 Deprecated stuff

In older versions of the `mmgroup` project the functionality of the `mm_op` extension was spread over several `Cython` extensions with names `mm`, `mm3`, `mm7`, `mm15`, etc. There are now python modules emulating the functionality of these deprecated extensions, which appear to work in Windows and Linux, but possibly not in macOS.

Note that the modules `mmgroup.mm`, `mmgroup.mm3`, `mmgroup.mm7`, etc., which emulate the old functionality, are also deprecated. So the user is **strongly** discouraged from using these deprecated modules. He or she should use the `mm_op` extension instead!

In older versions some python functions had to select the C function for the requested modulus, which has caused rather nasty problemes in some cases.

In the `mm_op` extension each documented `.c` function has been wrapped by a `Cython` function with the same name and the same signature. In the depreceated modules naming conventions are considerably more complicated.

3.6 Description of the `mmgroup.mm_reduce` extension

The functions in this module implement the fast reduction of an element of the monster group described in [Sey22]. There we define a triple of vector (v_1, v^+, v^-) in the representation ρ_{15} such that an element g of the monster van be recognized from the triple $(v_1 \cdot g, v^+ \cdot g, v^- \cdot g)$. A precomputed vector v_1 is stored in file `mm_order_vector.c`. Module `mm_order.c` contains functions for computing the order of an element of the monster. Module `mm_reduce.c` contains the function `mm_reduce_M` that implements the fast reduction algorithm in the monster group.

3.6.1 Generating an order vector

Python module `mmgroup.dev.mm_reduce.find_order_vector`

This module computes an order vector in a representation of the monster.

In [Sey22] we define a vector v_1 in the representation ρ_{15} of the monster group $\mathbb{M}$ that is used for recognizing elements of the subgroup G_{x0} of $\mathbb{M}$. Vector v_1 is also used for computing the order of an element of $\mathbb{M}$, so we call v_1 an **order vector** here.

v_1 is constructed from two vectors $v_{71} \in \rho_3$ and $v_{94} \in \rho_5$ via Chinese remaindering. This module contains functions for computing vectors v_{71} and v_{94} with the properties required in [Sey22].

These computations are the most time-consuming part of the computation of v_1; and we use multiprocessing for performing these computations.

All computations in the monster group are done with instances of class `MM0`, but not `MM`. The reason for this is that class `MM` requires the existence of an order vector.

3.6.2 Header file mm_reduce.h

Yet to be documented

Typedefs

typedef struct *gt_subword_s* **`gt_subword_type`**

typedef for structure `struct` *`gt_subword_s`*

typedef struct *gt_subword_buf_s* **`gt_subword_buf_type`**

typedef for structure `struct` *`gt_subword_buf_s`*

struct **`mm_compress_type`**

#include <mm_reduce.h> Structure for storing an element of the Monster compactly.

A properly *reduced* element of the Monster stored in a structure of type *`gt_word_type`* may also be encoded in this structure in a more compact form. This facilitates the conversion of that element to an integer, which is out of the scope of this module.

This structure may store an element of the Monster as a word of generators of shape

$$y_f \, x_d \, x_\delta \, \pi \, c_1 \, \tau_1 \, c_2 \, \tau_1 \, c_3 \, \tau_3 \, \ldots \, ,$$

where $d, f \in \mathcal{P}, \delta \in \mathcal{C}^*$, and $\pi \in \mathrm{Aut}_{\mathrm{St}} \mathcal{P}$. Here π must correspond to a generator with tag `p`. See section *Implementation of the generators of the monster group* in the *API reference* for details. τ_i is wqaul to generator τ or to its inverse.

A generator c_i is an element of the group G_{x0} referred by a 24-bit integer `c_i`. This must be one of the following:

If `c_i` represents a type-4 vector in *Leech lattice encoding* then this encodes the element of G_{x0} computed by applying the C function `gen_leech2_reduce_type4` to `c_i`.

If `c_i` represents a type-2 vector in *Leech lattice encoding* then this encodes the element of G_{x0} computed by applying the C function `gen_leech2_reduce_type2` to `c_i`.

The product $y_f \, x_d \, x_\delta \, \pi$ is encdoded in component `nx`, and the other generators are encdoded in the entries of component `w`, as desribed in the procedures below.

For background see section *Computations in the Leech lattice modulo 2* in *The mmgroup guide for developers*.

Public Members

uint64_t **`nx`**

encoding of $y_f\, x_d\, x_\delta\, \pi$

uint32_t **`w`**[MM_COMPRESS_TYPE_NENTRIES]

encoding of c_i, τ_i

uint32_t **`cur`**

index of last entry intered into component `w`

uint32_t **`back`**

True if structure is filled in reverse order.

struct **`gt_subword_s`**

#include <mm_reduce.h> Stucture to store a subword of generators of the Monster.

This structure is required in file `mm_shorten.c`.

It stores a subword of a word in the Monster in a node of a circular doubly-linked list. There is a dedicated EOF (end of file) mark in that list, in which member `eof`is set to 1; and elsewhere member `eof` is set to zero. Note that standard operations like insertions and deletions are easier in a circular doubly-linked list than in a standard doubly-linked list.

Member `data` contains a word g of generators of the subgroup G_{x0}; and member `t_exp` contains an exponent $0 \leq e < 3$. Then the structure represents the element $g\tau^e$ of the Monster, where τ is the triality element in the subgroup N_0 of the Monster.

The length of a word in member `data` is limited to the size `MAX_GT_WORD_DATA - 1`; and we will reduce that word using function `xsp2co1_reduce_word` in module `xsp2co1.c` if necessary. Note that member `data` may contain atoms with tags `'x', 'y', 'd', 'p', 'l'` only, and the inversion bit in such an atom is always cleared.

Member `img_Omega` contains the image $g \cdot \Omega$, where Ω is the standard frame of the Leech lattice mod 2. Here `img_Omega` is given in Leech lattice encoding. Member `reduced` is 1 if the word in member `data` is reduced (by function `xsp2co1_reduce_word`) and 0 otherwise.

Public Members

uint32_t **`eof`**

0 means standard subword, 1 means EOF mark

uint32_t **`length`**

Number of entries in component `data`

uint32_t **`img_Omega`**

Image of Ω under element.

uint32_t **`t_exp`**

Exponent of final tag ‘t’.

uint32_t **reduced**

True if part 'data' is reduced.

struct *gt_subword_s* ***p_prev**

Pointer to previous subword.

struct *gt_subword_s* ***p_next**

Pointer to previous subword.

uint32_t **data**[MAX_GT_WORD_DATA]

Element of monster group.

struct **gt_subword_buf_s**

#include <mm_reduce.h> Structure to store an array of entries of type `gt_subword_type`

We allocate several entries of of type `gt_subword_type` with a single call to function `malloc`. This saves a considerable amount of interaction with the operating system.

This structure contains an array of type `gt_subword_type[]` plus the necessary bookkeeping information.

Public Members

uint32_t **capacity**

max No of type `gt_subword_s` entries

uint32_t **n_used**

used No of type `gt_subword_s` entries

struct *gt_subword_buf_s* ***p_next**

Pointer to next buffer.

gt_subword_type **subwords**[1]

array of subwords in this buffer

struct **gt_word_type**

#include <mm_reduce.h> Stucture to store a word of generators of the Monster.

This structure is required in file `mm_shorten.c`.

It stores a word in the Monster in a circular doubly-linked list of nodes of type `gt_subword_type`. Each of these node represents subword of that word, and there is a dedicated EOF (end of file) mark in that list that marks both, the beginning and the end of the list. Member `p_end` always points to the EOF mark.

The structure contains a pointer `p_node` pointing to one of the nodes of the list, which we will call the **current** node. Some functions in these module take a pointer to this structure, and the perform operations on the current node. The pointer may be manipulated with function `gt_word_seek`.

Public Members

gt_subword_type ***p_end**

Pointer to the **end mark** subword.

gt_subword_type ***p_node**

Pointer to current subword.

gt_subword_type ***p_free**

Pointer to list of free subwords.

int32_t **reduce_mode**

Mode for the reduction of a word.

uint32_t **is_allocated**

1 if this structure has been allcocated

gt_subword_buf_type ***pb0**

pointer to first buffer

gt_subword_buf_type ***pbe**

pointer to last buffer

gt_subword_buf_type **buf**

1st buffer containing Array of subwords

3.6.3 C interface for file mm_order_vector.c

File `mm_order_vector` contains the precomuted `order_vector` and data related to that `order_vector`,

It also contains functions for retrieving these data.

Functions

int32_t **mm_order_load_tag_data**(uint32_t n, uint32_t *buf, uint32_t buf_size)

Load data from tables to a buffer.

The propose of this function is to provide the information for checking the correctness of the precomuted order vector v_1 stored in this module.

This function stores precomputed data in a buffer `buf` of type `uint32_t[buf_size]`, where `buf_size` is the size of the buffer. Parameter `n` specifies the value to be stored in the buffer. The function returns the length of the data in buffer. If the buffer is too short for that data then the function returns -1. A size parameter `buf_size = 128` is sufficient for all cases of `n`.

In case `n = 0` the function returns the array `TAG_VECTOR` of length 97. The tag vector is documented in file `mm_order.c`.

In case `n > 0` the function returns data required for verifying the precomputed order vector v_1. The order vector is discussed in [Sey22].

In case `n = 1,2,3` the function returns the values $g_{71} \in \mathbb{M}, v^0_{71} \in \rho_3, g \in \mathbb{M}$, respecively, in the internal sparse format. With these data we can compute:

$v_{71} = \sum_{i=0}^{70} v^0_{71} \cdot g^i_{71} g \pmod 3$.

In case `n = 4` the function returns the integer D in entry 0 of the buffer. In case `n = 5, 6` the function returns the values $g_{94} \in \mathbb{M}, v^0_{94}$, respectively. With this data we can compute:

$v_{94} = \sum_{i=0}^{93} (-1)^i \cdot v^0_{94} \cdot g^i_{94} \pmod 5$.

$v_1 = 10 \cdot v_{71} + 6 \cdot v_{94} + 5 \cdot D \cdot 1_\rho \pmod{15}$.

Here 1_ρ is defined as in [Sey22]. We can also check that the order vector v_1 satisfies the properties required in [Sey22].

In case `n = 7, 8, 9`, the function returnes the parts `TAGS_Y`, `TAGS_X`, and `TAG_SIGN`, respectively, of the `TAG_VECTOR`. For a description of the `TAG_VECTOR` see the description of `enum tag_offsets` in file `order_vector.c`.

void **`mm_order_load_vector`**(*uint_mmv_t* *p_dest)

Load order vector from tables to a buffer.

The function stores the precomputed order vector v_1 into the array referred by `p_dest`. That array must must be sufficiently long to store a vector of the representation ρ_{15}.

See [Sey22] for a decription of the properties of the vector v_1 in the representation ρ_{15} of the monster group.

The standard way to obtain the number of entries of type `unint_mmv_t` required for a vector of the representation ρ_{15} is to call function `mm_aux_mmv_size(15)` in file `mm_aux.c`.

int32_t **`mm_order_compare_vector`**(*uint_mmv_t* *p_v)

Compare vector with precomputed order vector.

The function compares the vector v in the representation ρ_{15} of the monster group referred by `p_v` with the precomputed order vector v_1.

The function returns 0 in case of equality and 1 otherwise.

int32_t **`mm_order_compare_vector_part_A`**(*uint_mmv_t* *p_v)

Compare A part of vector with precomputed order vector.

The function compares the A part the vector v in the representation ρ_{15} of the monster group referred by `p_v` with the A part of the precomputed order vector v_1.

The function returns 0 in case of equality and 1 otherwise.

uint64_t **`mm_order_hash_vector`**(*uint_mmv_t* *p_dest)

Return the hash value of the ordervector.

The function returns the hash value of the precomuted order vector as given by function `mm_aux_hash` in file `mm_aux.c`.

3.6.4 C interface for file mm_order.c

File `mm_order.c` contains functions for checking if an element is in the subgroup G_{x0} (or Q_{x0}) of the monster group. If this is the case, the element is expressed as a word in the generators of the corresponding subgroup.

File `mm_order.c` also contains functions for computing the order of an element of the monster.

All these function use the precomputed `order_vector` in file `mm_order_vector.c`

Enums

enum **`tag_offsets`**

The following enumeration contains the offsets of the (static) array `TAG_VECTOR` of unsigned 32-bit integers stored in this module. This array contains data that are required for dealing with the precomputed order vector v_1 stored in this module. That order vector can be obtained by calling function `mm_order_load_vector` in module `mm_order_vector.c`. The data in the array `TAG_VECTOR` can be obtained by calling function `mm_order_load_tag_data(0, buf, 97)` in the same module, with `buf` of type `uint32_t[97]`.

The purpose of the order vector v_1 is to identify an element g of the subgroup G_{x0} of the monster from $v_1 \cdot g$, as described in [Sey22]. The basic steps of this identification process are:

a. Given $v_1 \cdot g$, reduce g to an element g_1 of N_{x0}, see [Sey22] for background.
b. Given $v_1 \cdot g_1$, reduce $g_1 \in N_{x0}$ to g_2, where g_2 is in a certain 2 group of structure $2^{1+24+11}$. Therefore we have to watermark of the A part of $v_1 \cdot g_1$ with function `mm_op15_watermark_A` in module `mm15_op_eval_A.c`. Then we have to check that watermark against the precomputed watermark of the A part of v_1 using function `mm_op15_watermark_A_perm_num` in the same module. That precomputed watermark (of length 24) is stored in the array `TAG_VECTOR` at offset `OFS_WATERMARK_PERM`.
c. Given $v_1 \cdot g_2$, with g_2 as above, we want to present g_2 as a product of generators $g_2 = x_d x_\delta y_e$. For computing the factor y_e we have to extract (the signs of) 11 entries of $v_1 \cdot g_2$, with function `mm_aux_mmv_extract_sparse_signs` in module `mm_aux.c`. The coordinates and the expected values of these 11 entries are stored at the `TAG_VECTOR` at offset `OFS_TAGS_Y`. Once having extracted these sign bits we have to solve the linear equation system given by the 11 entries at the `TAG_VECTOR` at offset `OFS_SOLVE_Y`. This solution gives the value e of the factor y_e.
d. For computing the part $x_d x_\delta$ of g_2 (up to sign) we have to extract (the signs of) 24 more entries of the vector $v_1 \cdot g_2 y_e^{-1}$ and to solve another equation system as in the previous step. Here the relevant coordinates are stored at the `TAG_VECTOR` at offset `OFS_TAGS_X`, and the equation system is stored at offset `OFS_SOLVE_X`.
e. To correct the sign in the part $x_d x_\delta$, we have to extract another sign from $v_1 \cdot g_2 \cdot y_e^{-1} (x_d x_\delta)^{-1}$ at the coordinate given by `OFS_TAGS_SIGN`.

Values:

enumerator **`OFS_NORM_A`**

Sum of the squares of the A part of v_1 (mod 15)

enumerator **`OFS_DIAG_VA`**

For compatibly with older versions, always equal to 0

enumerator **`OFS_WATERMARK_PERM`**

Watermark of the A part of v_1 (mod 15) enumerator **OFS_TAGS_Y**

Entries of v_1 used for computing y_e

enumerator **OFS_SOLVE_Y**

Equation system used for computing y_e

enumerator **OFS_TAGS_X**

Entries of v_1 used for computing x_d

enumerator **OFS_SOLVE_X**

Equation system used for computing x_e

enumerator **OFS_TAG_SIGN**

Entry of v_1 used for computing sign of x_d

Functions

int32_t **mm_order_check_in_Gx0_fast**(*uint_mmv_t* *v)

Quick check if a vector is an image under an element of G_{x0}.

Assume that the vector v in ρ_{15} referred by `v` is the image of the precomputed order vector v_1 under an (unknown) element g of the monster. The function checks if that element g is in the subgroup G_{x0} of the monster.

The function may return 0, 1, or a negative value. If it returns 0 then g is not in G_{x0}. If it returns 1 then g is in G_{x0} with an error probability that is negligible for all practial purposes. A negative return value indicates an error.

int32_t **mm_order_check_in_Gx0**(*uint_mmv_t* *v, uint32_t *g, uint32_t mode, *uint_mmv_t* *work)

Check if a vector is an image under an element of G_{x0}.

Assume that the vector v in ρ_{15} referred by `v` is the image of the precomputed order vector v_1 under an (unknown) element g of the monster. We want to find that element g if it is in the subgroup G_{x0} of the monster.

We first describe the action of the function in the standard case that parameter `mode` is zero.

The function computes the element g if it is in the subgroup G_{x0} of the monster; otherwise it fails.

In case of success the function writes the element g into the array `g` as a word in the generators of G_{x0} of length at most 11; and the function returns the length of that word.

If no such element g_1 has been found then the function returns a number greater than 256. Then the exact return value gives some indication why no such element g has been found; this is for debugging only.

A negative return value indicates an internal error.

The function also requires an array `work` of length 15468 as a work buffer. This is the capacity required for storing a vector in the representation ρ_{15}.

Parameter `mode` is a bit field that modifies the action of the function as follows:

If bit 0 is set, then the inverse of the result g is returned in buffer `g`.

If bit 1 is set then we insert an atom into the word in `g` that indicates the image of the vector $\Omega \in \Lambda/2\Lambda$ under conjugation with g_0. Such an atom is considered as a comment, acting as the neutral element of the monster group. This is required for function `mm_reduce_M` in file `mm_reduce.c`. See documentation in that file for details.

If bit 2 is set then we perform an action to be used for internal tests only. Then the function computes an element g_1 such that $g^{-1}g_1 \in Q_{x0}$ holds in case $g \in G_{x0}$. If $g \notin G_{x0}$ then the function detects this fact with high probability. Usually, g_1 is a shorter word in the generators of the monster than g.

If bit 3 is set then input vector `v` is preserved; but the work buffer `work` must have the (doubled) length 30936 for storing an additional temporary copy of vector `v`.

int32_t **`mm_order_element_Gx0`**(uint32_t *g, uint32_t n, uint32_t *h, uint32_t o)

Compute exponent e such that $g^e \in G_{x0}$.

Let g be the element of the monster group stored in the array `g` as a word of generators of the monster group of length `n`.

The function computes the smallest exponent e such that g^e is in G_{x0}. Then the function writes $h = g^e$ into the buffer `h` as a word of generators of G_{x0} of length at most 10. Let k be the length of the word representing h. Then the function returns the value 0x100 $\cdot\, e + k$; here we have $1 \leq e \leq 119$ and $0 \leq k \leq 10$.

Computation of e is time consuming, and in some cases we are interested in small values of e only. Parameter `o` is an upper bound for the exponent e . The function may abort and return 0 in if e is greater than `o`; then the data in buffer `h` are invalid.

A negative return value indicates an internal error.

int32_t **`mm_order_element_M`**(uint32_t *g, uint32_t n, uint32_t o)

Compute order of an element g of the monster.

Let g be the element of the monster group stored in the array `g` as a word of generators of the monster group of length `n`.

The function returns the order of g.

Computation of the order is time consuming, and in some cases we are interested in small orders only. Parameter `o` is an upper bound for the order. The function may return 0 if the order is greater than `o`.

A negative return value indicates an error.

3.6.5 C interface for file mm_reduce.c

Function `mm_reduce_M` is the most important C function in this project. It reduces an arbitrary element of the monster group (represented as a word in the generators of the monster) to a word of fixed maximum length. Function `mm_reduce_M` uses the method in [Sey22] for reducing an element of the monster.

Here a word in the monster group is represented as an array of 32-bit integers of type `uint32_t` as described in section **The monster group** in the API reference. Such an array represents a word of generators of the monster. Thus group multiplication is concatenation of words. An generator in a word (given by an entry in the array of integers) can be inverted by flipping its most significant bit.

According to [Sey22], and element g of the monster group can be computed from the images v^+g, v^-g, and $v_1 g$. Here $v^+, v^- \in \rho_{15}$ are certain **2A axes**, and $v_1 \in \rho_{15}$ is certain vector used for recognizing an element of the subgroup G_{x0} of the monster, see ibid. for details. Representation ρ_{15} is the 198884-dimensional faithful representation of the monster group modulo 15. Vector v_1 is also called an **order vector**; and we can use the functions in module `mm_order.c` for dealing with a fixed precomputed order vector.

An element of ρ_{15} is implemented as a array of `MM_OP15_LEN_V` integers of type `uint_mmv_t` as described in **The C interface of the mmgroup project**, section **Description of the mmgroup.mm extension**. The value `MM_OP15_LEN_V` is defined in file `mm_op15.h`.

For reducing an element g of the monster we have to analyze the images v^+g, v^-g, and $v_1 g$ in that order. This file also exports subfunctions for performing these tasks. We use subfunctions `mm_reduce_vector_vp`, mm_reduce_vector_vm, and mm_reduce_vector_v1 for analyzing $v^+ g$, $v^- g$, and $v_1 g$, respectively. These functions accumulate their results in a buffer (of type uint32_t[] and size at least 128) that will eventually contain the reduced word equal to g.

The following code example shows an implementation of the main function mm_reduce_M based on these subfunctions, ignoring the error handling.

```
// The following buffers contain the element ``g`` to be reduced
uint32_t a[256];                // buffer containing element ``g``
uint32_t n;                     // length of data in buffer ``a``

// The folling buffers will contain the reduced element ``g``
uint32_t r[256];                // buffer for reduced element ``g``
uint32_t len;                   // length of data in buffer ``r``

// Temporary work buffers
uint_mmv_t v[MM_OP15_LEN_V];     // buffer for 2A axes
uint_mmv_twork[MM_OP15_LEN_V];   // work buffer

mm_op15_store_axis(V_PLUS, v );  // Store 2A axis V_PLUS in v
mm_op15_word(v, a, n, 1, work);  // multiply v_plus with g
mm_reduce_vector_vp(0, v, mode, r, work); // reduce V_PLUS * g

mm_op15_store_axis(V_MINUS, v );  // Store 2A axis V_MINUS in v
mm_op15_word(v, a, n, 1, work);  // multiply v_minus with g
mm_reduce_vector_vm(0,v, r, work); // reduce V_MINUS * g

mm_order_load_vector(v);         // load the order vector v_1
mm_op15_word(v, a, n, 1, work);  // multiply v_1 with g
len = mm_reduce_vector_v1(v, r, work);  // reduce v_1 vector

// Now buffer ``r`` contains the reduced word of length ``len``
// equal to the element ``g`` of the monster group.
```

Functions

uint32_t **mm_reduce_2A_axis_type**(*uint_mmv_t* *v)

Return type of a 2A axis in rep of monster.

Let v be a vector in the 196884-dimensional representation of the monster group modulo 15, encoded as described in section *Description of the mmgroup.mm<p> extensions* in the description of the *C interface*.

If v is a 2A axis then the function computes the type of the 2A axis. Each 2A axis corresponds uniquely to a 2A involution x in the monster. Let z be the central involution in the subgroup G_{x0} of the monster. Then the type of the 2A axis v is the class of the product xz in the monster, which has one of the values 2A, 2B, 4A, 4B, 4C, 6A, 6C, 6F, 8B, 10A, 10B, 12C

The function returns a value n * 2**28 + k * 2**24 + v. Here n is the number of the class, e.g. n == 6 for class 6C, k encodes the letter in the name of the class (‘A’ = 1, ‘B’ = 2, . . .) e.g. k = 3 for class 6C.

Output v is either 0 or a vector in the Leech lattice mod 2 found during the computation. Here a vector of type 2 is returned for classes 2A, 6A; and a vector of type 4 is returned for class 4A.

Caution:

This is a quick disambiguation of the type of a 2A axis. The function may return any axis type if `v` is not a 2A axis.

int32_t **`mm_reduce_analyze_2A_axis`**(*uint_mmv_t* *v, uint32_t *r)

Analyze a 2A axis in rep of monster.

Let `v` be a vector in the 196884-dimensional representation of the monster group modulo `15`, encoded as described in section *Description of the mmgroup.mm<p> extensions* in the description of the *C interface*.

If `v` is a 2A axis then the function analyzes the type of the 2A axis. Each 2A axis corresponds uniquely to a 2A involution x in the monster. Let z be the central involution in the subgroup G_{x0} of the monster. Then the type of the 2A axis `v` is the class of the product xz in the monster, which has one of the values 2A, 2B, 4A, 4B, 4C, 6A, 6C, 6F, 8B, 10A, 10B, 12C

The function returns 0 in case of success and a negative value value if it detects an error.

Buffer `r` should be at least 896 bytes long. It will contain the result.

Details are yet to be documented!!!

Caution:

This is a quick analyis of a 2A axis. The function returns garbage if `v` is not a 2A axis.

uint32_t **`mm_reduce_find_type4`**(uint32_t *v, uint32_t n, uint32_t v2)

Select a type-4 vector from list of Leech lattice vectors

Let `v` be an array of `n` vectors in the Leech lattice modulo 2 in Leech lattice encoding.

If parameter `v2` is 0 then the function returns the least vector of type 4 in the list `v` with repect to a certain ordering. If list `v` contains no vector of type 4 then the function returns 0.

Here subtypes are ordered as follows: `[48, 40, (42, 44), 46, 43]`. So e.g. a vector of subtype `40` is selected in favor of a vactor of subtype `46`. We do not distinguish between substypes `42` and `44`. From all vectors of the same subtype (equalizing subtypes `42` and `44`) we return the smallest vector in the list in the natural order.

If `v2` is nonzero then we accept only vectors `v1` in the list `v` with `type(v1) = 4` and `type(v1 + v2) = 2`.

The list `v` is destroyed. More precisely, all entries of `v` are masked with `0xffffff` and subjected to an (undocumented) permutation.

int32_t **`mm_reduce_transform_v4`**(*uint_mmv_t* *v, uint32_t v4, uint32_t *target_axes, uint32_t *r, *uint_mmv_t* *work)

Transform a 2A axis to a 'better' 2A axis.

The function tries to transform the 2A axis v given by parameter `v` to a better axis. Therefore it first applies a transformation $g \in G_{x0}$ given by parameter `v4`. Here `v4` must be a vector of type 4 in the Leech lattice mod 2 in *Leech lattice encoding*. Then g is the transformation mapping `v4` to the standard type 4-vector `Omega` as computed by function `gen_leech2_reduce_type4`.

In the next step this function tries to apply the transfomations τ^e for $e = 1, 2$ (in that order) until the type of the axis $vg\tau^e$ is in a set of types given by parameter `target_axes`. Here `target_axes` must be an array of length 2 containing the feasible axis types, encoded as described in function `mm_reduce_2A_axis_type`. An unused entry in array `target_axes` should be set to `0xffffffff`.

If a suitable transformed axis $vg\tau^e$ has been found then the function stores the element $g\tau^e$ in the array referred by `r`; and it returns the length of the data in the array `r`. Then it also overwrites the axis in buffer `v` with the transformed axis.

The function returns a negative value in case of any error, e.g. if no suitable transformed axis v has been found; then buffers `v` and `r` will contain garbage.

If the original axis v is orthogonal to the standard 2A axis then the transformed axis is also orthogonal to that axis.

Buffer `r` must have size at least 7. The function requires a work buffer `work` of the same type and size as parameter `v`.

int32_t **mm_reduce_load_axis**(*uint_mmv_t* *v, uint32_t s)

Load the 2A axis v^+ or v^- into a vector in ρ_{15}.

The function loads the 2A axis v^+ (in case `s == 0`) or v^- (in case `s == 1`) in the vector referred by `v`. Other values of `s` are illegal. Vector `v` will store an element of the representation ρ_{15} of the monster group as described in the header of this file. Vectors v^+ and v^- are defined in [Sey22], Section 6.

int32_t **mm_reduce_map_axis**(uint32_t *vt, *uint_mmv_t* *v, uint32_t *a, uint32_t n, *uint_mmv_t* *work)

Transform a 2A axis with an element of the Monster group.

Here we encode a 2A axis v as follows. It may be encoded as an element of the group Q_{x0} in the integer referred by `vt`. In case `*vt = 0` we encode v as a vector of the representation ρ_{15} of the Monster in the array `v`. In case `*vt != 0` the value `*vt` must be a short element (corresponding to a type-2 vector in $\Lambda/2\Lambda$) of the group Q_{x0}, in **Leech lattice encoding**. `v` is ignored in case `*vt != 0`.

Let g be the element of the monster given as a word of generators in the buffer `a` of length `n`. Then the function computes $v' = v \cdot g$ and stores v' in `*vt` (if possible), or in `v`. In last case it sets `*vt` to zero.

The function returns 0 in case of success an a negative value in case of failure. It requires a work buffer `work` of the same type and size as parameter `v`.

int32_t **mm_reduce_vector_vp**(uint32_t *vt, *uint_mmv_t* *v, uint32_t mode, uint32_t *r, *uint_mmv_t* *work)

Step 1 of the reduction of an element of the monster.

Here inputs `vt` and `v` of this function encode a 2A axis v as described in function `mm_reduce_map_axis`. Here v is considered as an image $v = v^+ \cdot g$ of v^+ under the operation of an (unknown) element g of the Monster group.

The function computes an element h of the monster group with $v \cdot h = v^+$. We use the method in [Sey22], Section 6 ff. for computing h.

The function stores h as a word of generators in the array `r` as described in the header of this file and returns the length of this word. That word is augmented by some atoms acting as comments

The buffer referred by `r` should have length at least 128.

Parameter `mode` is yet to be documented! In the current version is should be set to `1`. This means that the function will strictly follow the reduction method in ibid.

A negative return value indicates a fatal error.

Parameter `work` must refer to a work buffer of size at least `MM_OP15_LEN_V`.

More details are yet to be documented!!!!

int32_t **mm_reduce_vector_shortcut**(uint32_t stage, uint32_t mode, uint32_t axis, uint32_t *r)

Simulate steps of the reduction of an element of the monster.

Yet to be documented!!!!

int32_t **mm_reduce_vector_vm**(uint32_t *vt, *uint_mmv_t* *v, uint32_t *r, *uint_mmv_t* *work)

Step 2 of the reduction of an element of the monster.

This function should be applied immediately after function `mm_reduce_vector_vp`.

Here inputs `vt` and `v` of this function encode a 2A axis v as described in function `mm_reduce_map_axis`. Here v is considered as an image $v = v^+ \cdot g$ of v^+ under the operation of an (unknown) element g of the Monster group.

The function computes an element h of the monster group with $v \cdot h = v^-$. We use the method in [Sey22], Section 8, for computing h. Note that h preserves the 2A axis v^+.

The function appends h as a word of generators to the array `r` as described in the header of this file and returns the length of the data in the array `r`. That information in buffer `r` is augmented by some atoms acting as comments.

The buffer referred by `r` should have length at least 128.

A negative return value indicates a fatal error.

Parameter `work` must refer to a work buffer of size at least `MM_OP15_LEN_V`.

More details are yet to be documented!!!!

int32_t **`mm_reduce_vector_v1`**(*uint_mmv_t* *v, uint32_t *r, *uint_mmv_t* *work)

Step 3 of the reduction of an element of the monster.

This function should be applied immediately after functions `mm_reduce_vector_vp` and `mm_reduce_vector_vm`.

Here input `v` of this function must refer to the image of the precomputed order vector $v_1 \in \rho_{15}$ under the operation of the same element g of the monster group as described in the documentation of function `mm_reduce_vector_vp`.

The function computes an element h of the monster group with $v \cdot h = v_1$. We use the method in [Sey22], Appendix A. for computing h. Note that h preserves the 2A axes v^+ and v^-.

The function appends h as a word of generators to the array `r` as described in the header of this file and returns the length of the data in the array `r`. That information in buffer `r` is augmented by some atoms acting as comments.

Finally, the function computes a word in the buffer that is equal to the element g of the monster group and stores that element in the buffer `r`. The function returns the length of data in buffer `r`.

The buffer referred by `r` should have length at least 128.

A negative return value indicates a fatal error.

Parameter `work` must refer to a work buffer of size at least `MM_OP15_LEN_V`.

static inline int32_t **`mm_reduce_vector_v1_mod3`**(*uint_mmv_t* *v, uint32_t *r, *uint_mmv_t* *work)

Variant of Step 3 of the reduction of an element of the monster.

This is a variant of function `mm_reduce_vector_v1` that works with precomputed order vector $v_1 \in \rho_3$ instead of an order vector in ρ_{15}. Parameters and operation are the same as in function `mm_reduce_vector_v1`. But here input `v` of this function must refer to the image of the precomputed order vector $v_1 \in \rho_3$ under the operation of the same element g of the monster group as described in the documentation of function `mm_reduce_vector_vp`.

The order vector $v_1 \in \rho_3$ can be obtained by using the functions in module `mm_vector_v1_mod3.c`.

This function is faster than function `mm_reduce_vector_v1`. But the order vector in $v_1 \in \rho_3$ may have a non-trivial centralizer in the Monster; so we cannot use it for checking equality of elements of the Monster.

The buffer referred by `r` should have length at least 128.

A negative return value indicates a fatal error.

Parameter `work` must refer to a work buffer of size at least `MM_OP3_LEN_V`.

static inline int32_t **mm_reduce_vector_shorten**(uint32_t *a, uint32_t n, uint32_t *r, *uint_mmv_t* *work)

Special case of Step 3 of reduction of an element of the monster.

This is a variant of function `mm_reduce_vector_v1` that works in the special case when all prefixes of a word g in the Monster map the 2A-involutions v^+ and v^- to 2A-involutions in the subgroup Q_{x0} of G_{x0}. Here the mapping is done via conjuation. That condition holds e.g if all prefixes of the word g are in $\in N_x \cdot G_{x0}$.

The function returns 0 in case of success and a negative value in case of failure. In case of success, this function is (at least) 100 times faster than function `mm_reduce_vector_v1`. So using this function leads to a substantial speedup when computing in the 2-local subgroups N_x or G_{x0} of the Monster. The user profits from this speedup without effort, since function `mm_reduce_M` calls function `mm_reduce_vector_shorten` when appropriate.

Here the original element g of the Monster to be reduced must be stored in the buffer `a` of length `n` as usual. Parameters `r` and `work` are as in function `mm_reduce_vector_v1`.

If function `mm_reduce_vector_shorten` returns a negative value then we may retry the final reduction of g with one of the functions `mm_reduce_vector_v1` or `mm_reduce_vector_v1_mod3`.

int32_t **mm_reduce_vector_incomplete**(uint32_t *r)

Step finish incomplete reduction of an element of the monster.

This function should be applied immediately after function `mm_reduce_vector_vp` or after function `mm_reduce_vector_vn`.

The function outputs the word already stored in the buffer `r` computed by the functions `mm_reduce_vector_vp` and, possibly, `mm_reduce_vector_vn` and returns the length of the data in buffer `r`. Here we assume that subsequent calls to the functions in the sequence

`mm_reduce_vector_vp, mm_reduce_vector_vn, mm_reduce_vector_v1`,

that have not yet been called, would contribute nothing the the output assembled in buffer `r`.

The buffer referred by `r` should have length at least 128.

A negative return value indicates a fatal error.

More details are yet to be documented!!!!

int32_t **mm_reduce_M**(uint32_t *a, uint32_t n, uint32_t mode, uint32_t *r)

Reduce an element in the monster group.

This is the most important function in the project. It reduces an arbitrary element g of the monster to an element h. The value of the element h depends on the value of g as an element of the monster group, but not of the representation of g.

Let g be the element of the monster given as a word of generators in the buffer `a` of length `n`. Then the function computes the reduced element h of the monster with $g = h$. We use the method in [Sey22] for computing h.

The function stores h as a word of generators in the array `r` as described in the header of this file and returns the length of this word.

A negative return value indicates a fatal error.

The buffer referred by `r` should have length at least 128.

Parameter `mode` should usually be set to zero. It is interpreted as follows:

If bit 0 of `mode` is set then the reduction is strictly compatible with the reduction process described in [Sey22]. Otherwise we perform a more practical reduction. The latter reduction ensures e.g. that an element of the subgroup G_{x0} is always represented as a word in the generators of that subgroup. This feature greatly accelerates computations in the subgroup G_{x0}.

Bit 1 of `mode` should be set for tests only. If this bit is set then we omit some simplifications of the input prior to the actual reduction.

uint32_t **mm_reduce_set_order_vector_mod15**(uint32_t mode)

Set a special reduction mode for tests.

The end user need not call this function.

In [Sey22] we show how to compute an 'order vector' v_1 in the representation ρ_{15} of the Monster satisfying the following property:

The only element of the Monster fixing v_1 is the neutral element of the Monster.

For verifying this property of v_1 is suffices to use the methods in [LPWW98]. Thus by tracing the an image of v_1 under an element g back to its preimage v_1 we obtain a verification of a reduction process which depends on classical results only.

By default, the main reduction function `mm_reduce_M` in this module uses an 'order vector' in ρ_3 which leads to a faster reduction process than obtained by using v_1.

When calling function `mm_reduce_set_order_vector_mod15` with parameter `mode = 1` then in all subsequent calls to function `mm_reduce_M` we will use an 'order vector' v_1 in ρ_{15} as desrcibed above; and we will verify that the 'reduced' form of an element g of the Monster maps v_1 to the same vector as the original element g does.

This gives a tester the capability to do long calcuations in the Monster group that will be verified by using classical results without referring to the theory developed in [Sey22].

The user may switch back to the default behaviour by calling function `mm_reduce_set_order_vector_mod15` with parameter `mode = 0`. The function returns the value of parameter `mode` passed in the previous call (or the default value 0 at first call).

3.6.6 C interface for file mm_shorten.c

This file contains function for reducing a word of generators of the Monster group. Here word reduction is done by computations in the subgroups $G_{x0} = 2^{1+24}.\mathrm{Co}_1$ and $N_0 = 2^{2+11+22}.(\mathrm{Mat}_{24} \times \mathrm{Sym}_3)$. To some extent, we will also compute in the Frobenius group of order 21 generated by the generators τ and ξ of the Monster.

Internally, we store a word `a` of generators of the Monster in a structure of type *`gt_word_type`*. Here the word `a` is split into subwords, where each subword is in the double coset $G_{x0}N_0$. Such a subword is represented as a word of generators of G_{x0}, followed by a power of the triality element ξ. A subword is stored in a structure of type `gt_subword_type`.

Unless otherwise stated, a negative return value of a function is this module indicates an error.

Structures required for this module are defined in file `mm_reduce.h`.

The following code block demonstrates the reduction of a word with the functions in this module (ignoring error handling).

```
// Assume that the following buffer contains a word ``g``
// of length 10 of generators of the Monster group
#define INPUT_LENGTH 10
uint32_t in_buf[INPUT_LENGTH];
// Declare a sufficiently long output buffer
#define MAX_OUTPUT_LENGTH 20
uint32_t out_buf[MAX_OUTPUT_LENGTH];
// Define a reduction mode as specified in function gt_word_alloc
#define REDUCTION_MODE 1
// Allocate a structure for reduction
gt_word_type* p_gt = gt_word_alloc(REDUCTION_MODE, NULL, 0);
// Enter the word ``g`` into that structure
```

(continues on next page)

(continued from previous page)

```
gt_word_append(p_gt, in_buf, INPUT_LENGTH);
// Reduce the word in that structure
gt_word_reduce(p_gt);
// Store the reduced word in the output buffer and the actual
// length of that word in variable output_length
int32_t output_length;
output_length = gt_word_store(p_gt, out_buf, MAX_OUTPUT_LENGTH);
// Free the allocated structure
gt_word_free(p_gt);
```

Functions

void **gt_subword_clear**(*gt_subword_type* *p_gtsub)

Low-level function.

Set a subword to the empty subword.

gt_word_type ***gt_word_alloc**(uint32_t mode, void *p_buffer, size_t nbytes)

Create a structure of type `gt_word_type`

This function creates a valid structure of type `gt_word_type` and returns a pointer to that structure. The returned structure contains the empty word of generators of the Monster. If a buffer `p_buffer` of `nbytes` bytes length is given as a parameter then the structure may be ininitalized inside this buffer. We recommend `nbytes >= 4096`.

If `p_buffer = NULL` or `nbytes` is too small then a buffer is allocated via the C function `malloc`. You should always use the returned pointer as a pointer to a valid structure of type `gt_word_type` containing the neutral element of the Monster.

The structure referred by returned pointer will contain the circular doubly linked list with a single entry of type `gt_subword_type`. This entry is an end-of-file mark. So the whole word stored in that structure corresponds to the neutral element of the Monster group.

In any case function `gt_word_free` must be called to free all buffers allocated by this function and, possibly, by other function in this module applied to the pointer returned by this function.

Parameter `mode` specifies the reduction mode:

`mode = 0` means no reduction. Here the user has to apply the low-level functions in this module to perform reduction. This mode is used for testing.

`mode = 1` means the maximum possible reduction which is supported.

`mode = 2` means a lazy mode of reduction. The idea is to reduce only in cases where we can expect a substantial decrease of the word length. This mode is faster than `mode = 1`.

This function returns `NULL` in case of failure, e.g. if the C function `malloc`\`fails to allocate memory.

void **gt_word_free**(*gt_word_type* *p_gt)

Free a structure of type `gt_word_type`

This function frees the memory allocated to a structure of type `gt_word_type` created by function `gt_word_alloc`. Ultimately, it must be applied to all pointers created by function `gt_word_alloc` in order to avoid memory leaks. It is safe to apply function `gt_word_free` to the `NULL` pointer.

void **gt_word_clear**(*gt_word_type* *p_gt)

Set content of the structure `*p_gt` to the empty word.

This function sets the value of the structure of type *gt_word_type* referred by `p_gt` to the neutral element of the Monster. It also frees all allocated memory blocks referred by `p_gt`, except for the block `p_gt` itself.

int32_t **gt_word_insert**(*gt_word_type* *p_gt)

Low-level function.

Insert an empty subword after the current subword pf `*p_gt` and set the position of the current subword to the new subword.

Return a negative value if out of memory.

int32_t **gt_word_delete**(*gt_word_type* *p_gt)

Low-level function.

Delete the current subword, and set the position of the current subword to its predecessor.

Return a negative value on an attempt to delete an EOF mark.

int32_t **gt_word_seek**(*gt_word_type* *p_gt, int32_t pos, uint32_t set)

Low-level function.

Move the pointer to the current subword to position 'pos' Her pos = 0 means the EOF mark if 'set' is True, and the position of the current subword otherwise. pos = i+1 means the subword after pos = i, pos = i-1 means node subword pos = i.

It is an error to move p_gt->p_node across the EOF node. When starting at the EOF node we may move forward or back until finding the EOF node once again, but not any further.

int32_t **gt_word_append_sub_part**(*gt_word_type* *p_gt, uint32_t *a, uint32_t n)

Low-level function.

Try to append a prefix of word 'a' of length 'n' to the current subword. Return number of atoms of 'a' sucessfully appended to that word. A negative return value inducates an error.

int32_t **gt_word_reduce_sub**(*gt_word_type* *p_gt, uint32_t sub_mode)

Low-level function.

Reduce current subword

sub_mode = 0: lazy reduction (just some word shortening)

sub_mode = 1: standard reduction

sub_mode = 2: as sub_mode 0, but move also prefix in N_x0 to previous subword

sub_mode = 3: as sub_mode 1, but move also prefix in N_x0 to previous subword

int32_t **gt_word_rule_join**(*gt_word_type* *p_gt)

Low-level reduction function.

Try to join current subwort `p_gt->p_node` with its predecessor. Return 1 and let `p_gt->p_node` point to first node being changed if the rule could be applied. Return 0 and do not change `p_gt->p_node` this is not the case. Return a negative number in case of a fatal error.

int32_t **gt_word_rule_t_xi_t**(*gt_word_type* *p_gt)

Low-level reduction function.

This function tries to apply a reduction rule:

$\tau^{e_1}\xi^{e_2}\tau^{e_3} \rightarrow \xi^{e_4}\tau^{e_5}\xi^{e_6}$.

Such a rule exists for all $0 < e_1, e_2, e_3 < 3$. This rule is applied to the subword `p_gt->p_node` and to its predecessor. Applying such a rule decreases the number of generators $\tau^{\pm 1}$ in a word.

The function returns 1 and lets `p_gt->p_node` point to first node being changed if such rule could be applied. It returns 0 and does not change `p_gt->p_node` this is not the case. It returns a negative number in case of a fatal error.

To show the existence of suitable reduction rules as mentioned above we put $a = \tau\xi^{-1}$. Then we verify that a has order 7 and that $\tau^{-1}a\tau = a^2$ holds. Therefore is suffices to verify that the group operations corresponding to these relations fix the 'order vector' in the representation ρ_{15} .

Since τ has order 3, the group generated by τ and a (and hence also the group generated by τ and ξ) is visibly a Frobenius group of order 21. Finding suitable reduction rules in such a group is easy.

int32_t **`gt_word_append`**(*gt_word_type* *p_gt, uint32_t *a, uint32_t n)

Append word of generators to strcture `*p_gt`

Append the word `a` of length `n` to the word in the buffer refered by `p_gt`.

A negative return value inducates an error.

At exit, pointer `p_gt->p_node` will point to the `EOF` mark.

uint32_t **`gt_word_n_subwords`**(uint32_t *a, uint32_t n)

Return number of subwords required to store element of the Monster.

Return (an upper bound for) the number of subwords in a doubly linked list of type *`gt_word_type`* required to store the word of generators of the Monster of length `n` referred by `a`.

int32_t **`gt_word_length`**(*gt_word_type* *p_gt)

Return the length of the word stored in `*p_gt`.

int32_t **`gt_word_reduce`**(*gt_word_type* *p_gt)

Reduce the word stored in `*p_gt`.

The function reduces the word g stored in the structure `*p_gt`. The mode of reduction depends on the parameter `mode` passed to function `gt_word_alloc` when creating that structure. The function returns:

0 if g is not known to be in any subgroup of the Monster

4 if g is known to be in $G_{x0} \cdot N_0$

5 if g is known to be in N_0

6 if g is known to be in G_{x0}

7 if g is known to be in $N_{x0} = G_{x0} \cap N_0$

A negative value in case of failure.

If the return value is > 0 (i.e. if we know $g \in G_{x0} \cdot N_0$) then we also have performed a full reduction of the result.

int32_t **`gt_word_store`**(*gt_word_type* *p_gt, uint32_t *pa, uint32_t maxlen)

Store the word in `*p_gt` in an array.

The function stores the word in `*p_gt` in the array `a` (of length `maxlen`) referred by `pa`. It returns the actual length `n`of the word stored in the array `a`.

The function returns a negative value in case of failure, e.g. if `n > maxlen`.

int32_t **`gt_word_to_mm_compress`**(*gt_word_type* *p_gt, *mm_compress_type* *p_c)

Compress word in `*p_gt` to strcuture of type *`mm_compress_type`*

Yet to be tested and documented!!!

int32_t **`gt_word_compress`**(*gt_word_type* *p_gt, uint64_t *p_n)

Compress word in `*p_gt` to a 256-bit integer``.

Yet to be tested and documented!!!

int32_t **`gt_word_shorten`**(uint32_t *g, uint32_t n, uint32_t *g1, uint32_t n1max, uint32_t mode)

Reduce a word of generators of the Monster group.

Let g be the word of the monster group of length `n` stored in the array `g`. The function reduces the word g and stores the reduced word in the buffer `g1` of length `n1max`. It returns the actual length `n1` of the reduced word.

Internally, the function uses a buffer of type *`gt_word_type`* for reduction. Parameter `mode` controls the reduction mode; it is specified as in function `gt_word_alloc`.

The function returns a negative value in case of failure, e.g. if `n1 > n1max`.

3.6.7 C interface for file mm_compress.c

This file contains functions for compressing a word of generators of the Monster group to an integer.

TODO: Documentation yet missing!

Functions

void **`mm_compress_pc_init`**(*mm_compress_type* *pc, uint32_t back)

yet to be documented

int32_t **`mm_compress_pc_add_nx`**(*mm_compress_type* *pc, uint32_t *m, uint32_t len)

yet to be documented

int32_t **`mm_compress_pc_add_type2`**(*mm_compress_type* *pc, uint32_t c)

yet to be documented

int32_t **`mm_compress_pc_add_type4`**(*mm_compress_type* *pc, uint32_t c)

yet to be documented

int32_t **`mm_compress_pc_add_t`**(*mm_compress_type* *pc, uint32_t t)

yet to be documented

int32_t **`mm_compress_pc`**(*mm_compress_type* *pc, uint64_t *p_n)

yet to be documented

int32_t **`mm_compress_pc_expand_int`**(uint64_t *p_n, uint32_t *m)

yet to be documented

3.6.8 C interface for file mm_vector_v1_mod3.c

File `mm_vector_v1_mod3` contains a precomuted vector `v1_mod3` of the representation of the monster group (mod 3). This vector can be used for obtaining an (unknown) element g of the subgroup G_{x0} of the monster from and image of that vector under g.

So the functions in this module are similar to those in module `mm_order.c`, but faster.

Note that we cannot check membership in G_{x0} with the functions in this module!

This module is yet a stub!!!

Functions

void **`mm_order_load_vector_v1_mod3`**(*uint_mmv_t* *p_dest)

Load order vector from tables to a buffer.

The function stores the precomputed vector `v1_mod3` into the array referred by `p_dest`. That array must must be sufficiently long to store a vector of the representation ρ_3.

int32_t **`mm_order_compare_v1_mod3`**(*uint_mmv_t* *v)

Compare vector with precomputed order vector.

The function compares the vector v in the representation ρ_3 of the monster group referred by `v` with v_1 , where v_1 is the precomute vector `v1_mod3`.

The function returns 0 in case of equality and 1 otherwise. It destroys the vector in the buffer `v`.

int32_t **`mm_order_find_Gx0_via_v1_mod3`**(*uint_mmv_t* *v, uint32_t *g)

Find an element g of the subgroup G_{x0}.

Yet to be documented!!!

3.7 Shared libraries and dependencies between Cython extensions

For each Cython extension the relevant C functions are implemented in a shared library (or a DLL in Windows). Names of the corresponding DLLs in windows are given in the following table:

Table 5: Shared libraries implementing Cython extensions

Cython extension	Implemented in shared library
`mm_reduce`	`mmgroup_reduce`
`mm_op`	`mmgroup_mm_op`
`clifford12`	`mmgroup_mat24`
`generators`	`mmgroup_mat24`
`mat24`	`mmgroup_mat24`

In Linux, the name of each shared library is prefixed by the string `lib`. So the Windows DLL `mmgroup_mm_op.dll` corresponds to the Linux shared library `libmmgroup_mm_op.so.`

Cython extensions depend on each other as shown in the following paragraph:

```
mm_reduce
   + mm_op
      + clifford12
         + generators
             + mat24
```

CHAPTER

FOUR

APPLICATIONS OF THE MMGROUP PACKAGE

4.1 Introduction

Some applications using the *mmgroup* package can be downloaded from the github repository

https://github.com/Martin-Seysen/mmgroup .

In that repository each application is stored in a subdirectory of directory `applications`.

The *mmgroup* package must be installed before running any of these applications, as described in the **API reference** of this project.

4.2 Representing the Monster as a Hurwitz group

We want to find a presentation of the Monster as a Hurwitz group.

4.2.1 Description of the task to be done

Wilson [Wil01] has shown that the Monster group is a Hurwitz group, i.e. a group generated by two elements a, b satisfying the relations $a^2 = b^3 = (ab)^7 = 1$.

In a **MathOverflow** question is has been asked for the smallest possible order of the commutator $[a, b] = a^{-1}b^{-1}ab$ of a and b in such a presentation of the Monster, see

https://mathoverflow.net/q/363882 .

Using the implementation of the Monster group in the *mmgroup* project, it is now possible to find a non-trivial upper bound for that commutator. The purpose of this application is to find generators a, b of the Monster satisfying the relations given above, so that we can compute the order of the commutator $[a, b]$.

In [Wil01] Wilson has given explicit generators a (in class 2B) and b (in class 3B) of the Monster satifying the relations given above. It appears to be extremely difficult to translate these two generators from the language used in [Wil01] to the language used in our *mmgroup* package. Since not all such pairs a, b must be isomorphic (via conjugation), we also want to find several such pairs. So we have to reimplement the search for such pairs here.

At the time of writing this document we have found 7 pairs of such generators a, b; and in both cases the commutator $[a, b]$ has order 39. I thank Gabriel Goller in Augsburg, Germany for computing most of these pairs.

Python scripts referred in the following subsections can be found in subdirectory `applications/Hurwitz` of the repository.

4.2.2 Searching for pairs of generators

Let G_{x0} be the subgroup of the Monster of structure $2^{1+24}.\mathrm{Co}_1$ as defined in the **API reference** of this package, and let z_{-1} be the central involution in that subgroup. Then for generating a pair a, b as above, we put $a = z_{-1}$. Then we search for a random $g_3 \in G_{x0}$ of order 3 with $\chi_1(g_3) = 53$, so that g_3 is in class 3B of the Monster. Here χ_1 is the character of degree 196883 of the Monster. With the *mmgroup* package we can compute that character in much less than a second. For a fixed g_3 we generate a large number of (sufficiently) random elements g_e of the Monster and we put $g = g_e^{-1} g_3 g_e$. If zg has order 7 then $(a, b) = (z, g)$ is a candidate for a pair of generators, and we store g_3 and g_e in a file for further processing.

The python script `find_generators.py` searches for pairs (a, b) as above, and stores the corresponding elements g_3, g_e in file `Hurwitz_order7.txt`. Here for checking the order of ab we simply check that $v \cdot (ab)^7 = v$ for a random vector v in the 196883-dimensional representation of the Monster modulo 3. This is by far the most time-consuming computation in the whole application. Wilson [Wil01] reports a success rate of one over 64 millions for finding a suitable pair. Using all CPUs of the authors's computer, we can check about 12 million candidates per day, so that our search has to run for about a week.

4.2.3 Verifying pairs of generators

Given a pair (a, b) satisfying the relations $a^2 = b^3 = (ab)^7 = 1$, we also have to verify that a and b actually generate the Monster group and not just a proper subgroup of the Monster. Here the strategy in [Wil01] is to compute random elements in the group $\langle a, b \rangle$ generated by a and b, until one finds an element h_1 of order 94 and an element h_2 or order 41, 59, or 71. According to [Wil01], every proper subgroup of the Monster containing an element of order 94 is contained in a maximal subgroup of structure $2.B$, where B is the Baby Monster. But none of the primes 41, 59, or 71 divides the order of B; so h_1 and h_2 generate the Monster.

We compute a certificate for checking that a and b generate the Monster as follows. Put $H(a, b, n) = \prod_{i=0}^{63} ab^{1+\nu_i}$ for elements a, b of the Monster, and an integer $0 \leq n < 2^{64}$ with binary representation $n = \sum_{i=0}^{63} \nu_i \cdot 2^i, \nu_i \in \{0, 1\}$. If a and b generate the Monster then we can effectively find integers n_1, n_2 such that $H(a, b, n_1)$ has order 94 and $H(a, b, n_2)$ has order 41, 59, or 71.

The script `parse_found.py` reads the file `Hurwitz_order7.txt` discussed in the previous section and checks if pair of candidates in that file actually generates the Monster. If this is the case for a pair then it stores that pair plus a certificate for that pair in a list in file `hurwitz_monster_samples.py`. Computing such a certificate take a few minutes for a pair (a, b) on the author's computer, which is negligible compared to the effort for finding such a pair.

4.2.4 Using pairs of generators

For using a pair of generators of the Monster, the user may simply import the file `hurwitz_monster_samples.py`. This file essentially contains a list of pairs of elements of the Monster, so that each of these pairs generates the Monster as a Hurwitz group. That list also contains certificates for these pairs as discussed in the last section. A precomputed file `hurwitz_monster_samples.py` containing (at least) two such pairs is stored in the `git` repository of the project.

The script `hurwitz_verify.py` checks all pairs (a, b) contained in file `hurwitz_monster_samples.py` and displays the order of the commutator $[a, b]$. This activity takes just a few seconds for each pair.

4.3 Mapping the Coxeter group Y_{555} into the Bimonster

Sample programs dealing with the group Y_{555} and the Bimonster

The implementation of the Coxeter group Y_{555} and the Bimonster has now been integrated into the *mmgroup* project. Documentation see *The Coxeter group Y_{555} and the Bimonster* in the **API reference**.

This directory now just contains a few tiny self-explaining applications of the implementation of Y_{555} and the BiMonster.

4.4 Computing standard generators of the Monster

This application computes standard generators of the Monster.

The standard generators `(a, b)` of the Monster satisfy the following properties:

`a` is in class 2A, `b` is in class 3B and `ab` has order 29,

see

https://brauer.maths.qmul.ac.uk/Atlas/v3/spor/M/

Function `standard_generators` in file `stdgen.py` in this application computes standard generators of the Monster (if this has not yet been done) and stores them in the file `standard_generators.py`.

CHAPTER

FIVE

DEMONSTRATION CODE FOR THE REDUCTION ALGORITHM

This chapter has been written for a mathematically-inclined reader who wants to read an implementation demonstrating the reduction algorithm for the Monster group in [Sey22]. In this section we present a Python implementation of that reduction algorithm. This implementation has been optimized for readability, and not for speed; but it can still be executed and tested.

Our goal is present a satisfactory demonstration of the new reduction algorithm in this chapter of the project documentation. Executable Python code can be found in the **mmgoup.demo** package. Function **reduce_monster_element** in the **mmgroup.demo.reduce_monster** package actually reduces an element of the Monster.

Demonstrating every tiny detail of the reduction algorithm in Python leads to a bloated software package that is hardly readable or executable. So we have to make a compromise, which parts of the reduction algorithm we want to demonstrate.

Following the *Pacific island model* in [Wil13], we assume that computations in the subgroup G_{x0} of the Monster (of structure $2^{1+24}.\mathrm{Co}_1$) are easy enough, so that demonstration code is not needed for them. The relevant computations in G_{x0} are described in detail in the appendices of [Sey22].

We also assume that functions for the operation of the Monster on its 196884-dimensional representation ρ_{15} (with coefficients taken modulo 15) are available. This operation is described in detail in [Sey20]. In the **mmgroup** package we use multiplication for that operation.

We also assume that functions performing linear algebra with matrices over the Leech lattice (modulo 2 and 3) are available.

Section *Subfunctions for the reduction algorithm* describes the interface of functions implementing the tasks mentioned above.

Warning: Functions and classes presented in Chapter *Demonstration code for the reduction algorithm* are for demonstration only. They should not be used in a life application!

This demonstration makes some random choices in the reduction process. This is appropriate for testing. The life implementation of the reduction algorithm is deterministic.

5.1 Data structures

This section contains the data structures required for the demonstration of the reduction algorithm. These data structures are classes that can be imported from module **mmgroup.demo**.

`class mmgroup.demo.Mm(tag=None, atom=None, *args, **kwds)`

Models an element of the Monster group

Here it would be tempting to use the class *MM* defined in the *API reference* of the **mmgroup** package for computing in the Monster instead. Note that methods of class *MM* may apply the reduction algorithm whenever appropriate. Hence it would be cheating to demonstrate the reduction algorithm with instances of that class.

So we use this class **Mm** for demonstrating the reduction algorithm for the Monster.

The constructor of the class **Mm** performs the same action as the constructor of class *MM*. The generators of the Monster used here are discussed in [Sey22], Section 3 - 5.

But multiplication of instances of class **Mm** is simply a concatenation of words, without any attempt to reduce the result.

We use the following special cases of calling the constructor:

Calling **Mm('r', k)** constructs a random word in the generators of the Monster containing **k** triality elements $\tau^{\pm 1}$, for an integer **k**.

Calling the constructor with the string **'negate_beta'** constructs an element of the normal subgroup Q_{x0} of the subgroup G_{x0} of the Monster that exchanges the axes v^+ and v^- defined in [Sey22], Section 7.2.

`property count_triality_elements`

Length of a word representing an element of the Monster

The function returns the number of the triality elements $\tau^{\pm 1}$ contained in the word representing the group element. This will be used as an indication for the length of the word. This is similar to (although not equal to) the definition of the length of a word in [Wil13].

`class mmgroup.demo.MmV15(vector_name=0)`

Models a vector in the representation of the Monster mod 15

The constructor constructs a vector in the representation of the Monster mod 15 depending on the parameter **vector_name** passed as an argument. If **vector_name** is the string 'v+' or 'v-' then we construct the vector v^+ or v^- defined in [Sey22], Section 7.2, respectively.

If **vector_name** is the string 'v1' then the we construct a precomputed vector v_1 satisfying the requirements in [Sey22], Section 6. The computation of such a vector v_1 is discussed in [Sey22], Appendix B.

Two such vectors in this class may be checked for equality. A vector may be multiplied with an element of the Monster group. Other operations are not supported.

`class mmgroup.demo.Leech2(value)`

Models an vector in the Leech lattice mod 2

Such vectors may be added. A vector may be multiplied with an element of the subgroup G_{x0} of the monster.

Calling the constructor with the string 'Omega' or 'beta' returns the vector λ_Ω or λ_β defined in [Sey22], Section 4.1 or 7.1, respectively.

Calling the constructor with an integer argument returns the element of the Leech lattice mod 2 with that number. Here the numbering is as in class *XLeech2*, setting the sign bit to zero.

property type

Return type of vector in Leech lattice mod 2.

The type of a vector is the halved norm of a shortest preimage of the vector in the Leech lattice. That type is equal to 0, 2, 3, or 4.

5.2 The reduction algorithm for the Monster group

This module demonstrates the reduction algorithm for the Monster

The main function **reduce_monster_element** in this module reduces an element of the Monster group. Given an word g of generators of the Monster of an arbitrary length, the function returns an word h of generators of the Monster of length at most 7, such that $g \cdot h$ is the neutral element of the Monster. Here the length of a word is is the number of triality elements $\tau^{\pm 1}$ contained in that word.

Module **mmgroup.demo.reduce_monster** requires the following external references.

```
from mmgroup.demo import Mm, Leech2, MmV15
from mmgroup.demo.reduce_sub import *
from mmgroup.demo.reduce_axis import reduce_axis
from mmgroup.demo.reduce_feasible import reduce_feasible_axis
```

Here comes the main function **reduce_monster_element** of the module that reduces an element of the Monster group.

```
def reduce_monster_element(g):
    r"""Reduce an element g of the Monster

    :param g: Element of the Monster of arbitrary length
    :type g:  class Mm
    :return: Element h of length at most 7 with g * h == 1
    :rtype:  class Mm
    """
    # Compute h such that g * h is in the subgroup H^+ of the Monster
    v_plus = MmV15('v+') * g
    h = reduce_axis(v_plus)                   # Now g * h is in H^+
    # Compute h such that g * h is in the subgroup H of G_x0
    v_minus = MmV15('v-') * g * h
    h = h * reduce_feasible_axis(v_minus)     # Now g * h is in H
    # Reduce the element g * h in the group G_x0
    v_1 = MmV15('v1') * g * h
    h = h * reduce_G_x0(v_1)                  # Now g * h is 1
    return h
```

Here is a simple test function for testing the main function **reduce_monster_element**.

```
def check_reduce_monster_element():
    r"""Generate a random element g of the Monster and test reduction of g
    """
    g = Mm('r', 14)                           # Random Monster element g
    assert g.count_triality_elements == 14    # Check length of g
    h = reduce_monster_element(g)             # Reduce g; the result is h
    assert MmV15('v1') * g * h == MmV15('v1') # Check that g * h is 1
    assert h.count_triality_elements <= 7     # Length of h must be <= 7
```

The following function reduces an element of the subgroup G_{x0} of the Monster. The implementation of this function is discussed in [Sey22], Section 6.

```
def reduce_G_x0(v):
    r"""Reduce an element g of the subgroup G_x0 of the Monster

    :param v: Vector in representation of the Monster, see below
    :type v: class MmV15
    :return: Element g of G_x0 as defined below
    :rtype:  class Mm

    Let v be a vector in the representation of the Monster
    satisfying the condidition

    v * g = MmV15('v1')

    for an unknown element g of the group G_x0. The function computes
    g if such an element g exists. Otherwise it raises an error.
    """
    v2 = v.copy()                          # Local copy of v

    # Compute kernel of the matrix corresponding to part 300_x of v;
    # that kernel must contain a unique Leech lattice vector l2 of type 4
    r3, l2 = mat15_rank_3(v2, 0)
    assert r3 == 23                        # Rank of 300_x must be 23

    # Compute element g1 of G_x0 that transforms l2 to \lambda_\Omega
    g1 = map_type4_to_Omega(l2)
    v2 = v2 * g1                           # Transform v2 with g1

    # Now v2 * g2 = MmV15('v1') for a (uinque) element g2 of the group N_x0
    g2 = find_in_Nx0(v2)                   # Finding g2 is easy
    assert v * g1 * g2 ==  MmV15('v1')     # This is what we expect
    return g1 * g2
```

5.3 Reducing an axis in the Monster

Module **mmgroup.demo.reduce_axis** demonstrates the reduction of an axis

Here the *axes* are certain vectors in the representation ρ_{15} the Monster. The axes are in a one-to-one correspondence with the left cosets of the subgroup H^+ of structure $2.B$ of the Monster. Thus mapping an arbitrary axis **v** to the standard axis v^+ corresponding to the subgroup H^+ is equivalent to mapping an element of the Monster to an element of H^+ by right multiplication. For background we refer to [Sey22], Section 7.

The process of finding an element g that maps axis **v** to v^+ is called *reduction* of the axis **v**. Function **reduce_axis** in this module reduces an axis **v**.

We name an orbit of an axis under the action of the subgroup G_{x0} by a string as in [Sey22], Section 8.2. In the sequel an *orbit* of an axis means an orbit under the action of G_{x0}. Function **axis_orbit** in this module computes the orbit of an axis.

For reducing an axis we first multiply an axis with a suitable element of G_{x0} in order to obtain a 'nice' axis in the same orbit. Roughly speaking, an axis is 'nice' if multiplication with a suitable power of the triality element τ of the Monster maps that axis into a 'simpler' orbit.

This way we may repeatedly multiply an axis first with an element of G_{x0} and then with a power of τ, leading to a 'simpler' orbit in each step of the reduction process, as discussed in [Sey22], Section 8. Possible sequences of orbits obtained during such a reduction process are shown in Figure 2 in [Sey22], Section 8.3.

For each axis $\mathbf{v}$ there is a set $U_4(\mathbf{v})$ of vectors in the Leech lattice mod 2 such that for any $g \in G_{x0}$ the axis $\mathbf{v}g$ is 'nice', if there is an $l_2 \in U_4(\mathbf{v})$ with $l_2 g = \lambda_\Omega$. For background we refer to [Sey22], Section 8.3. Function **compute_U** in this module is used to compute the set $U_4(\mathbf{v})$. More precisely, calling **compute_U(v)** returns a set $U(\mathbf{v})$ of vectors in the Leech lattice mod 2; and $U_4(\mathbf{v})$ is the set of type-4 vectors contained in that set. Function **map_type4_to_Omega** in module **mmgroup.demo.reduce_sub** computes g from l_2.

At the end of that process we obtain an axis in the orbit **'2A'**. That orbit also contains the axis v^+. Mapping an arbitrary axis in orbit **'2A'** to the standard axis v^+ is easy.

Module **mmgroup.demo.reduce_axis** requires the following external references.

```
from random import choice                      # returns a random entry of a list
from mmgroup.demo import Mm, Leech2, MmV15   # data strucures used
from mmgroup.demo.reduce_sub import *        # functions used
```

Here is the implementation of function **reduce_axis**.

```
def reduce_axis(v):
    r"""Return element of Monster reducing an axis v to the standard axis v^+

    :param v: The axis to be reduced
    :type v: class MmV15
    :return: Element g of the Monster with v * g = v^+
    :rtype: class Mm

    Here v^+ is the standard axis.
    """
    v1 = v.copy()                   # Local copy of axis v
    g = Mm(1)                       # Neutral element of the Monster

    # In g we will accumulate the element of the Monster that transforms v

    # Map axis to a 'simpler' orbit; see documentation of the module
    while True:
        orbit = axis_orbit(v1)      # Orbit of current axis v
        if orbit == '2A':
            break                   # Done if we are in orbit '2A'

        # Compute the set U_4(v) and select a random element l2 of that set
        U = compute_U(v1)
        U_4 = [l2 for l2 in U if l2.type == 4]
        l2 = choice(U_4)            # A random element of U_4(v)

        # Find a Monster element g1 that maps v1 to a 'nice' axis
        # and map v1 to that 'nice' axis
        g1 = map_type4_to_Omega(l2)
        v1 = v1 * g1                # Transform v1 with g1
        g = g * g1
        assert v * g == v1          # Correctness condition for loop

        # Find a Monster element g_tau that maps v1 to a 'simpler' axis,
```

(continues on next page)

(continued from previous page)

```
        # and map v1 to that 'simpler' axis
        target_orbits = TARGET_ORBITS[orbit] # possible 'simpler' orbits
        g_tau = find_triality_element_for_axis(v1, target_orbits)
        v1 = v1 * g_tau             # Transform v1 with g_tau
        g = g * g_tau
        assert v * g == v1          # Correctness condition for loop

    # Now axis v has been transformed to an axis v1 in orbit '2A'.
    # Map the short Leech lattice vector l2 corresponding to axis v1
    # to the standard short vector \lambda_\beta.
    _, l2 = mat15_rank_3(v1, 2)
    g1 = map_type2_to_standard(l2) # g1 transforms l2 to \lambda_\beta
    v1 = v1 * g1                   # Transform v1 with g1
    g = g * g1

    # Here Here v1 must be v^+ or v^-
    assert v1 in [MmV15('v+'), MmV15('v-')]
    if v1 != MmV15('v+'):
        G_MINUS = Mm('negate_beta')
        v1 = v1 * G_MINUS           # Correct v1 if it is equal to v^-
        g = g * G_MINUS
    assert v * g == MmV15('v+')    # This is what we expect
    return g
```

Dictionary **TARGET_ORBITS** encodes the graph shown in Figure 2 in [Sey22], Section 8.3. The vertices of that graph are orbits of axes.

```
TARGET_ORBITS = {
  '2B' : ['2A'],
  '4A' : ['2A'],
  '4B' : ['2B'],
  '4C' : ['2B'],
  '6A' : ['4A'],
  '6C' : ['4A'],
  '6F' : ['4C'],
  '8B' : ['4A'],
  '10A' : ['6A'],
  '10B' : ['4B', '4C'],
  '12C' : ['4B', '6A'],
 }
```

Here is the implementation of function **axis_orbit**. This function has been implemented according to the discussion of the orbits of axes in [Sey22], Section 8.4.

```
def axis_orbit(v):
    """Compute the orbit of an axis

    :param v: The axis to be checked
    :type v: class MmV15
    :return: Name of the orbit of axis v
    :rtype: str
    """
```

(continues on next page)

(continued from previous page)

```
    ORBITS_KNOWN_FROM_NORM = {
        2:'8B', 3:'4C', 5:'6C', 10:'12C', 13:'4B'
    }
    assert isinstance(v, MmV15) and v.p == 15

    # Compute norm of the 24 times 24 matrix 300_x contained in vector v (mod 15)
    norm = mat15_norm(v)

    if norm in ORBITS_KNOWN_FROM_NORM:
        # If there is only one orbit with that norm, return that orbit
        return ORBITS_KNOWN_FROM_NORM[norm]
    elif norm == 4:
        # If 300_x has norm 4 then compute the rank r3 of matrix
        # M = 300_x - 2 * M_1, where M_1 is the unit matrix
        r3, l2  = mat15_rank_3(v, 2)
        if r3 == 23:
            # If M has rank 23 then we check that the kernel of M
            # contains a vector l2 of type 2 in Leech lattice
            if l2.type == 2:
                # Compute y = transposed(l2) * M * l2 (mod 15)
                y = mat15_apply(v, l2)
                if y == 4:
                    return '2A'  # Orbit is '2A' if y == 4
                elif y == 7:
                    return '6A'  # Orbit is '6A' if y == 7
                else:
                    raise ValueError("Vector is not an axis")
            else:
               raise ValueError("Vector is not an axis")
        elif r3 == 2:
            return '10A'         # Orbit is '10A' if M has rank 2
        else:
            raise ValueError("Vector is not an axis")
    elif norm == 8:
        # If 300_x has norm 8 then compute rank r3 of matrix 300_x
        r3, _,  = mat15_rank_3(v, 0)
        if r3 == 8:
            return '2B'          # Orbit is '2A' if rank is 8
        elif r3 == 24:
            return '10B'         # Orbit is '10B' if rank is 24
        else:
            raise ValueError("Vector is not an axis")
    elif norm == 14:
        # If 300_x has norm 14 then compute rank r3 of matrix 300_x
        r3, _, = mat15_rank_3(v, 0)
        if r3 == 8:
            return '6F'          # Orbit is '6F' if rank is 8
        elif r3 == 23:
            return '4A'          # Orbit is '4A' if rank is 23
        else:
            raise ValueError("Vector is not an axis")
    else:
```

(continues on next page)

(continued from previous page)

```
        raise ValueError("Vector is not an axis")
```

Here is the implementation of function **compute_U**. For an axis **v** the function computes the set **U(v)** of vectors in the Leech lattice (mod 2) defined in [Sey22], Section 8.4. The implementation of the function is essentially a rewrite of the discussion of the different cases of axis orbits in that section.

```
def compute_U(v):
    """Compute a set of Leech lattice vectors from an axis v

     For a given axis v the function computes the set U(v) of
     vectors in the Leech lattice mod 2, as defined above.

    :param v: The axis v
    :type v: class MmV15
    :return: The set U(v) of vectors in the Leech lattice (mod 2)
    :rtype: list[Leech2]
    """
    assert isinstance(v, MmV15) and v.p == 15
    orbit = axis_orbit(v)        # Compute the orbit of axis v
    if orbit == '2A':
        return []
    if orbit == '2B':
        return leech2_span(vect15_S(v, 4))
    if orbit == '4A':
        _, l2  = mat15_rank_3(v, 0)
        return [l2]
    if orbit in ['4B','4C']:
        return leech2_rad(vect15_S(v, 1))
    if orbit == '6A':
        _, l2  = mat15_rank_3(v, 2)
        return [x + l2 for x in vect15_S(v, 5)]
    if orbit == '6C':
        return leech2_span(vect15_S(v, 3))
    if orbit in ['6F', '12C']:
        return leech2_rad(vect15_S(v, 7))
    if orbit == '8B':
        S1 = (vect15_S(v, 1))
        l2 = choice(S1)          # l2 is a random element of S1
        return [l2 + x for x in S1]
    if orbit == '10A':
        S1 = (vect15_S(v, 1))
        S3 = (vect15_S(v, 3))
        l2 = S3[0]               # S3 is a singleton here
        return [l2 + x for x in S1]
    if orbit == '10B':
        return leech2_rad(vect15_S(v, 4))
    raise ValueError('Unknown orbit')
```

5.4 Reducing a feasible axis in the Baby Monster

Module **mmgroup.demo.reduce_feasible**: reduction of a feasible axis

Here the *feasible axes* are certain axes in the representation ρ_{15} of the Monster, see [Sey22], Section 9.1 for details. They are in a one-to-one correspondence with the left cosets of the subgroup $H = G_{x0} \cap H^+$ in the group H^+ defined in [Sey22].

Thus mapping an arbitrary feasible axis **v** to the standard feasible axis v^- corresponding to the subgroup H is equivalent to mapping an element of H^+ to an element of H by right multiplication.

The process of finding an element g in H^+ that maps a feasible axis **v** to v^- is called *reduction* of the feasible axis **v**. Function **reduce_feasible_axis** in this module reduces a feasible axis **v**.

The algorithm in function **reduce_feasible_axis** for reducing a feasible axis is discussed in [Sey22], Section 9. This algorithm is quite similar to the algorithm in function **reduce_axis** for reducing a general axis. Analogies between these two algorithms are summarized in Table 2 in that section.

An orbit of H^+ under the action of H will be called a H-orbit. As in function **reduce_axis**, we repeatedly multiply a feasible axis first with an element of H and then with a power of τ. This way we will map the axis into a 'simpler' H-orbit in each step of the reduction process.

Details of that reduction process are discussed in [Sey22], Section 9. Possible sequences of H-orbits obtained during such a reduction process are shown in Figure 3 in that section.

An orbit of the Monster under the action of G_{x0} will be called a G-orbit. Note that disjoint H-orbits lie in disjoint G-orbits, with the following exceptions. H-orbits **'2A0'** and **'2A1'** are in G-orbit **'2A'**; and H-orbits **'2B0'** and **'2B1'** are in G-orbit **'2B'**.

The reduction process mentioned above terminates if the feasible axis is in the H-orbit **'2A0'** or **'2A1'**. Orbit **'2A1'** is the singleton $\{v^-\}$, so that we are done in that case. Transforming a feasible axis from orbit **'2A0'** to orbit **'2A1'** is easy.

Module **mmgroup.demo.reduce_feasible** requires the following external references.

```python
from mmgroup.demo import Mm, Leech2, MmV15
from mmgroup.demo.reduce_axis import axis_orbit
from mmgroup.demo.reduce_axis import compute_U
from mmgroup.demo.reduce_axis import TARGET_ORBITS
from mmgroup.demo.reduce_sub import*
```

Here is the implementation of function **reduce_feasible_axis**.

```python
def reduce_feasible_axis(v):
    r"""Return element of Monster reducing a feasible axis v

    Here reducing a feasible axis means reduction to the
    standard feasible axis v^-.

    :param v: The feasible axis to be reduced
    :type v: class MmV15
    :return: Element g of subgroup H^+ of the Monster with v * g = v^-
    :rtype: class Mm
    """
    BETA = Leech2('beta')    # Vector \lambda_\beta in leech lattice mod 2
    OMEGA = Leech2('Omega')  # Vector \lambda_\Omega in leech lattice mod 2
    v1 = v.copy()            # Local copy of the feasible axis v
```

(continues on next page)

(continued from previous page)

```
    g = Mm(1)                # Neutral element of the Monster

    # In g we will accumulate the element of H^+ that transforms v

    # Map axis to a 'simpler' orbit
    while True:
        orbit = axis_orbit(v1)
        if orbit == '2A':    # Done if we are in orbit '2A1' or '2A0'
            break

        # Compute the set U_f(v), as defined in [Sey22], Section 9.2;
        # and select a random element l2 of that set
        U = compute_U(v1)
        U_f = [
            l2 + BETA for l2 in U if
            l2.type == 4 and (l2 + BETA).type == 2
        ]
        l2 = choice(U_f)     # A random element of U_f(v)


        # Find a Monster element g1 that maps v1 to a 'nice' axis
        # and map v1 to that 'nice' (feasible) axis
        g1 = map_feasible_type2_to_standard(l2)
        v1 = v1 * g1         # Transform v1 with g1
        g = g * g1
        assert v * g == v1   # Correctness condition for loop

        # Find a Monster element g_tau that maps v1 to a 'simpler' axis,
        # and map v1 to that 'simpler' (feasible) axis
        target_orbits = TARGET_ORBITS[orbit]
        g_tau = find_triality_element_for_axis(v1, target_orbits)
        v1 = v1 * g_tau      # Transform v1 with g_tau
        g = g * g_tau
        assert v * g == v1   # Correctness condition for loop

    # Now v has been transformed to an axis v1 in orbit '2A0' or '2A1'.
    # Compute the short Leech lattice vector l2 = \lambda(ax(v2)).
    _, l2 = mat15_rank_3(v1, 2)

    if l2 == BETA:
        # If l2 is \lambda_beta then v1 is in orbit '2A1' and we are done
        assert v * g == MmV15('v-')       # This is what we expect
        return g

    # Map v1 to an axis with \lambda(ax(v1)) = \lambda\beta + \lambda\Omega
    g1 = map_feasible_type2_to_standard(l2)
    g = g * g1
    v1 = v1 * g1             # Transform v1 with g1

    # Now a power of the triality element maps v1 to the
    # standard feasible axis v^-
    for e in [1, -1]:
```

(continues on next page)

(continued from previous page)

```
        v2 = v1 * Mm('t', e)
        if v2 == MmV15('v-'):
            g = g * Mm('t', e)
            assert v * g == MmV15('v-')   # This is what we expect
            return g

    raise ValueError("Vector is not a feasible axis")
```

5.5 Subfunctions for the reduction algorithm

The Python module **mmgroup.demo.reduce_sub** contains auxiliary functions required for the reduction on an element of the Monster group.

Most functions in this module correspond to functions defined in [Sey22]. Cross references are given in the documentation of a function.

Implementations of the functions in this module are usually just Python wrappers for the corresponding C functions in the **mmgroup** package.

`mmgroup.demo.reduce_sub.`**`mat15_norm`**`(`*v*`)`

Compute norm of a matrix related to a vector in ρ_{15}

Parameters

v (*class MmV15*) – vector in representation ρ_{15} of the Monster

Let A be the 24 times 24 matrix corresponding to part 300_x of vector **v**. The function returns the norm of the matrix A (reduced mod 15).

Returns

Norm of matrix A (mod 15)

Return type

int

This corresponds to computing $||M(v)|| \pmod{15}$ in [Sey22], Section 8.2.

`mmgroup.demo.reduce_sub.`**`mat15_apply`**`(`*v*, *l2*`)`

Apply matrix related to a vector in ρ_{15} to Leech lattice vector

Parameters

- **v** (*class MmV15*) – vector in representation ρ_{15} of the Monster
- **l2** (*class Leech2*) – vector in the Leech lattice mod 2 of type 2

Let A be the 24 times 24 matrix corresponding to part 300_x of vector **v**.

The function returns $l_2 A l_2^\top$ (reduced mod 15), where l_2 is any preimage of **l2** in the Leech lattice.

Returns

$l_2 A l_2^\top \pmod{15}$

Return type

int

This corresponds to computing $M(v, l_2) \pmod{15}$ in [Sey22], Section 8.3.

`mmgroup.demo.reduce_sub.mat15_rank_3(v, k)`

Compute rank of a matrix related to a vector in ρ_{15}

Parameters

- **v** (*class MmV15*) – vector in representation ρ_{15} of the Monster
- **k** (*int*) – scalar factor for the 24 times 24 unit matrix

Let A be the 24 times 24 matrix corresponding to part 300_x of vector **v**. Let A_k be the matrix A minus **k** times the unit matrix, with coefficients taken modulo 3. Matrix A_k has a natural interpretation as a matrix acting on the Leech lattice (mod 3).

Returns
tuple(**r, l2**) with **r** the rank of the matrix A_k

Return type
tuple(int, class Leech2)

If the kernel of A_k is one-dimensional and its preimage in the Leech lattice contains a nonzero vector **l3** of type at most 4 then we let **l2** be the (unique) vector in the Leech lattice mod 2 with **l2 = l3** (mod 2). Otherwise we put **l2 = None**.

This corresponds to computing the pair $(\mathrm{rank}_3(v,k), l_2)$, as defined in [Sey22], Section 8.3, with $l_2 \in \mathrm{ker}_3(v,k)$ if the conditions mentioned above are satisfied.

`mmgroup.demo.reduce_sub.vect15_S(v, k)`

Compute Leech lattice vectors related to a vector in ρ_{15}

Parameters

- **v** (*class MmV15*) – vector in representation ρ_{15} of the Monster
- **k** (*int*) – a number 0 < **k** < 8

The basis vectors of part 98280_x of **v** are in one-to-one correspondence with the shortest nonzero vectors of the Leech lattice, up to sign. The function returns the list of short Leech lattice vectors such that the corresponding co-ordinate of (part 98280_x of) **v** has absolute value **k** (modulo 15).

Returns
List of short Leech lattice vectors as described above

Return type
list[class Leech2]

This corresponds to computing $S_k(v)$ in [Sey22], Section 8.3.

`mmgroup.demo.reduce_sub.leech2_span(l2_list)`

List vectors in a subspace of the Leech lattice modulo 2

Parameters
l2_list (*list[class Leech2]*) – List of vectors in the Leech lattice mod 2

The function returns the list of all vectors in the Leech lattice mod 2 that are in the subspace spanned by the vectors in the list **l2_list**.

Returns
List of short Leech lattice vectors as described above

Return type
list[class Leech2]

This corresponds to computing the set $\mathrm{span}(S)$, as defined in [Sey22], Section 8.3, where S is the set of vectors given by the list **l2_list**.

`mmgroup.demo.reduce_sub.`**`leech2_rad`**`(`*`l2_list`*`)`

List vectors in a subspace of the Leech lattice modulo 2

Parameters
`l2_list` (*`list[class Leech2]`*) – List of vectors in the Leech lattice mod 2

The function returns the list of all vectors in the Leech lattice mod 2 that are in the radical of the subspace spanned by the vectors in the list **l2_list**. The radical of a subspace is the intersection of that subspace with its orthogonal complement.

Returns
List of short Leech lattice vectors as described above

Return type
list[class Leech2]

This corresponds to computing the set rad(S), as defined in [Sey22], Section 8.3, where S is the set of vectors given by the list **l2_list**.

`mmgroup.demo.reduce_sub.`**`map_type4_to_Omega`**`(`*`l2`*`)`

Map a type-4 vector in the Leech lattice mod 2 to λ_Ω

Parameters
`l2` (*`class Leech2`*) – Vector of type 4 in the Leech lattice mod 2

The function returns an element g of the group G_{x0} that maps the vector **l2** of type 4 in the Leech lattice mod 2 to the standard type-4 vector λ_Ω.

Returns
group element g mapping **l2** to λ_Ω

Return type
class Mm

An implementation of this function is discussed in [Sey22], Appendix A.

`mmgroup.demo.reduce_sub.`**`map_type2_to_standard`**`(`*`l2`*`)`

Map a type-2 vector in Leech lattice mod 2 to λ_β

Parameters
`l2` (*`class Leech2`*) – Vector of type 2 in the Leech lattice mod 2

The function returns an element g of the group G_{x0} that maps the vector **l2** of type 2 in the Leech lattice mod 2 to the standard type-2 vector λ_β.

Returns
group element g mapping **l2** to λ_β

Return type
class Mm

An implementation of this function is discussed in [Sey22], Appendix C.

`mmgroup.demo.reduce_sub.`**`map_feasible_type2_to_standard`**`(`*`l2`*`)`

Map a feasible type-2 vector in the Leech lattice mod 2 to $\lambda_\beta + \lambda_\Omega$

Parameters
`l2` (*`class Leech2`*) – Feasible vector of type 2 in the Leech lattice mod 2

The function returns an element g of the group G_{x0} that maps the *feasible* vector **l2** of type 2 in the Leech lattice mod 2 to the standard feasible type-2 vector $\lambda_\Omega + \lambda_\beta$. Here the term *feasible* is defined in [Sey22], Section 9.1.

Returns
group element g mapping **l2** to $\lambda_\beta + \lambda_\Omega$

Return type
class Mm

An implementation of this function is discussed in [Sey22], Appendix D.

`mmgroup.demo.reduce_sub.`**`find_triality_element_for_axis`**(*v*, *axis_orbits*)

Try to transform an axis in ρ_{15} into a given axis orbit

Parameters

- **v** (*class MmV15*) – an axis in the representation ρ_{15} of the Monster
- **axis_orbits** (*list[str]*) – List of expected types of the transformed axis

Let v be the axis given by **v**. The function computes the axes $v \cdot \tau^e$ for $e = \pm 1$, where τ is the triality element in the Monster.

If possible, the function returns an element τ^e, $e = \pm 1$ such that the type of the axis $v \cdot \tau^e$ is in the list **axis_orbits** of axis types. Names of axis types are as in [Sey22], Section 8.2.

The function raises **ValueError** if this is not possible. It does not change the axis stored in **v**.

Returns
An element τ^e of the Monster as described above

Return type
class Mm

Using function **axis_orbit** in module **mmgroup.demo.reduce_axis** this function can be implemented as follows:

```python
def find_triality_element_for_axis(v, axis_orbits)
    for e in [1, -1]:
        if axis_orbit(v * Mm('t', e)) in axis_orbits:
            return  Mm('t', e)
    raise ValueError
```

The implementation used here is much faster, since it computes fewer co-ordinates of the transformed axes.

`mmgroup.demo.reduce_sub.`**`find_in_Nx0`**(*v*)

Identify an element of the subgroup N_{x0} of the Monster

Parameters
v (*class MmV15*) – vector in representation ρ_{15} of the Monster

Let v_1 be the precomputed vector in ρ_{15} defined in the constructor of class *MmV15*. Vector **v** must be an image of v_1 under an element of the subgroup N_{x0} of the Monster. Then the function returns an element g of N_{x0} with $\mathbf{v}g = v_1$, if such an element g exists. Otherwise it raises **ValueError**. Note that at most one element g of the Monster may satisfy that condition.

Returns
An element g of the Monster as described above

Return type
class Mm

An implementation of this identification of an element of the group N_{x0} is described in [Sey22], Section 6. The precomputed vector v_1 satisfies the conditions stated in the same section.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[AG04] S. Aaronson and D. Gottesman. Improved simulation of stabilizer circuits. *CoRR*, 2004. URL: http://arxiv.org/abs/quant-ph/0406196.

[Asc86] M. Aschbacher. *Sporadic Groups*. Cambridge University Press, New York, 1986.

[Con85] J. H. Conway. A simple construction of the Fischer-Griess monster group. *Inventiones Mathematicae*, 79:513–540, 1985.

[CCN+85] J. H. Conway, R. T. Curtis, S. P. Norton, R. A. Parker, and R. A. Wilson. *Atlas of Finite Groups*. Clarendon Press, Oxford, 1985.

[CS99] J. H. Conway and N. J. A. Sloane. *Sphere Packings, Lattices and Groups*. Springer-Verlag, New York, 3rd edition, 1999. ISBN 0-387-98585-9.

[CNS88] J.H. Conway, S.P. Norton, and L.H. Soicher. The bimonster, the group y_555, and the projective plane of order 3. In *Computers in Algebra (Chicago 1985)*, 27–50. Lecture Notes in Pure and Appl. Math, 111, Dekker, New York, 1988.

[Far12] A. Farooq. Some computations in the monster group and related topics. Doctoral thesis, School of Mathematical Sciences. Queen Mary University of London, 2012. URL: https://qmro.qmul.ac.uk/xmlui/bitstream/handle/123456789/8484/Farooq_A_PhD_final.pdf.

[Gri82] R. L. Griess. The friendly giant. *Inventiones Mathematicae*, 69:1–102, 1982.

[Iva99] A. A. Ivanov. *Geometry of Sporadic Groups I, Petersen and Tilde Geometries*. Cambridge University Press, June 1999. ISBN 0521413621.

[Iva09] A. A. Ivanov. *The Monster Group and Majorana Involutions*. Cambridge University Press, April 2009. ISBN 0521889944.

[Iva92] A.A. Ivanov. A geometric characterization of the monster. In *Groups, combinatorics & Geometry (Durham, 1990)*, 46–62. London Math Soc. Lecture Notes Ser 165, Cambridge University Press, 1992.

[LPWW98] S. Linton, R. Parker, P. Walsh, and R. Wilson. Computer construction of the Monster. *J. Group Theory*, 1:307–337, 1998.

[NRS01] G. Nebe, E. M. Rains, and N. J. A. Sloane. The invariants of the clifford groups. *IJDCC: Designs, Codes and Cryptography*, 2001.

[Nor02] S. Norton. Transforming the monster presentation. Preprint, published as an appendix in [Far12], 2002. URL: https://qmro.qmul.ac.uk/xmlui/bitstream/handle/123456789/8484/Farooq_A_PhD_final.pdf.

[Nor98] S. P. Norton. Anatomy of the monster: i. In *The Atlas of Finite Groups - Ten Years On*, 198–214. Cambridge University Press, 1998.

[Nor92] S.P Norton. Constructing the monster. In *Groups, combinatorics & Geometry (Durham, 1990)*, 63–76. London Math Soc. Lecture Notes Ser 165, Cambridge University Press, 1992.

[Sey20] M. Seysen. A computer-friendly construction of the monster. *arXiv e-prints*, pages arXiv:2002.10921, February 2020. arXiv:2002.10921.

[Sey22] M. Seysen. A fast implementation of the Monster group. *arXiv e-prints*, pages arXiv:2203.04223, 2022. arXiv:2203.04223.

[Wik23] Wikipedia. Monster group. 2023. URL: https://en.wikipedia.org/wiki/Monster_group.

[Wil01] R. A. Wilson. The Monster is a Hurwitz group. *J. Group Theory*, 4:367–374, 2001.

[Wil13] R. A. Wilson. The Monster and black-box groups. *arXiv e-prints*, pages arXiv:1310.5016, October 2013. arXiv:1310.5016.

PYTHON MODULE INDEX

INDEX