Coding of a unitary representation of the Rubik's group

Sebastiano Corli, Enrico Prati

August 4, 2022

Abstract

The code implements a new way to represent the Rubik's Cube as a vector, and the elements from Rubik's group as unitary operators acting on it. Compositions between two or more group elements are implemented, returning the composed operator as output. Such implementation makes the program suitable for Machine Learning applications or algebraic simulations. The mathematical formalism beneath is detailed in the paper attached to the project. The software consists of the enhanced environment which was developed in the paper. Beyond Quantum Mechanics, the new environment can be used to simulate a Rubik's Cube, acting with the standard moves to rotate the layers and checking whether its configuration is the solved one or not. All the code has been revised and tested thanks to the unittest library, available for python 3.9.

1 Introduction

The code we present is developed in python 3.9 and is object-oriented. The program is articulated in three files: baseline.py, Rubik.py and unittest.py. In the first file, all the basic structures are provided. In the second file, two classes are defined, virtually the Rubik's Cube and the Rubik's group. The first class implements a vector which describes the state of the Cube, the second one the transformations on it. Transformations and states can be opportunely combined to set the game of the Cube. In the last file, every function and method provided by these classes are tested. The purpose of this program lies in implementing the unitary representation developed in the paper attached to the project, which is also available at this link, or alternatively at this one. In the first section, a basic usage of the code is illustrated, in the second section we examine how the algebraic rules have been implemented by the python code.

2 User guide

2.1 RubiksCube and RubiksGroup classes

All the code for now we need can be imported from Rubik.py file. In the first place, we shall see how to instantiate the vector which represents the state of the Rubik's Cube:

Source Code 2.1: Setting C object

```
1 from Rubik import RubiksCube
2 
3 ## Implement the Cube ##
4 Cube = RubiksCube()
5 print("-"*20)
6 print(Cube)
7 print("-"*20)
```

The class RubiksCube, defined in Rubik.py (line 1), allows to allocate in memory a vector state, which in fact represent the Rubik's Cube. When a RubiksCube object is initialized (line 4), the configuration returned by default is the solved one. The vector which expresses the solved state is given by the output:

```
--------------------
(0, 0, 0, [1. 0.])
(0, 0, 0, [1. 0.])
(0, 0, 0, [1. 0.])
(0, 0, 0, [1. 0.])
(0, 0, 0, [1. 0.])
(0, 0, 0, [1. 0.])
(0, 0, 0, [1. 0.])
(0, 0, 0, [1. 0.])
(0, 0, 0, [1. 0.])
(0, 0, 0, [1. 0.])
(0, 0, 0, [1. 0.])
(0, 0, 0, [1. 0.])
(0, 0, 0, [0. 1. 0.])
(0, 0, 0, [0. 1. 0.])
(0, 0, 0, [0. 1. 0.])
(0, 0, 0, [0. 1. 0.])
(0, 0, 0, [0. 1. 0.])
(0, 0, 0, [0. 1. 0.])
(0, 0, 0, [0. 1. 0.])
(0, 0, 0, [0. 1. 0.])
--------------------
```

There are twelve rows of zeros, followed by a [1. 0.] vector. Such rows represent the so called edges, i.e. the little cubes (or cubies) which display only two faces in the Cube. The three numbers stand for how much each cubie is far from its solved position, when all of these are zeros, the cubie lies in its solved one. The vector [1. 0.] means how the two faces of the edge cubie are arranged. The [1. 0.] vector matches the solved configuration, [0. 1.] the flipped one.
On the other hand, the Cube is built also by eight corner cubies, which instead present three faces. The triple of numbers, given by (0, 0, 0) in the output above, stands again for the distance of the corner from its solved cell, while the [0. 1. 0.] for the orientation of the cubie. [0. 1. 0.] corresponds to the solved orientation, [1. 0. 0.] for the anti-clockwise orientation and [0. 0. 1.] for the clockwise.
In the next lines, once the environment for the Cube is set, perform two transformations, say the rotation of the front layer (F) and the back one (B):

Source Code 2.2: F and B transformations on C

```python
1  from Rubik import RubiksCube, RubiksGroup
2  
3  ## Allocate a RubiksCube object ##
4  Cube = RubiksCube()
5  print("-"*20)
6  print(Cube)
7  print("-"*20)
8  
9  ## Allocate the front and back rotations ##
10 F = RubiksGroup.F()
11 B = Rubiksgroup.B()
12 
13 ## Act on the Rubik's Cube ##
14 F * Cube
15 B * Cube
16 print(Cube)
17 print("-"*20)
```

F and B operators are initialized by a class method (line 9, 10) which returns a specific object of the same class. To act such transformations on the state Cube, the * (`__mul__`) operator has been overwritten. The * operation supports the action of any generator from RubiksGroup on a RubiksCube object.

In the output, we can check the structure of Cube object before and after the two transformations:

```
--------------------
(0, 0, 0, [1. 0.])
(0, 0, 0, [1. 0.])
(0, 0, 0, [1. 0.])
(0, 0, 0, [1. 0.])
(0, 0, 0, [1. 0.])
(0, 0, 0, [1. 0.])
(0, 0, 0, [1. 0.])
(0, 0, 0, [1. 0.])
(0, 0, 0, [1. 0.])
(0, 0, 0, [1. 0.])
(0, 0, 0, [1. 0.])
(0, 0, 0, [1. 0.])
(0, 0, 0, [0. 1. 0.])
(0, 0, 0, [0. 1. 0.])
(0, 0, 0, [0. 1. 0.])
(0, 0, 0, [0. 1. 0.])
(0, 0, 0, [0. 1. 0.])
(0, 0, 0, [0. 1. 0.])
(0, 0, 0, [0. 1. 0.])
(0, 0, 0, [0. 1. 0.])
--------------------
(1, 0, -1, [0. 1.])
(0, 0, 0, [1. 0.])
(-1, 0, -1, [0. 1.])
(0, 0, 0, [1. 0.])
(-1, 0, 1, [0. 1.])
(0, 0, 0, [1. 0.])
(1, 0, 1, [0. 1.])
(0, 0, 0, [1. 0.])
(-1, 0, -1, [0. 1.])
(-1, 0, 1, [0. 1.])
(1, 0, 1, [0. 1.])
(1, 0, -1, [0. 1.])
(1, 0, 0, [1. 0. 0.])
(0, 0, -1, [0. 0. 1.])
(-1, 0, 0, [1. 0. 0.])
(0, 0, -1, [0. 0. 1.])
(0, 0, 1, [0. 0. 1.])
(-1, 0, 0, [1. 0. 0.])
(0, 0, 1, [0. 0. 1.])
(1, 0, 0, [1. 0. 0.])
--------------------
```

After the second dashed line, it is possible to see the new vector state, transformed by the F and B operators. Almost all the tuples of zeros have been substituted by 1 and -1, as well as many orientations, i.e. the [1. 0.] and [0. 1. 0.] vectors, have shifted to new configurations. Take notice, applying B and F or vice versa does not mean any difference, as far as B and F transformations commute each other with. However, such situation does not hold in general, but for the antipodal rotations (up and down, left and right, front and back).
At every line, the status of the RubiksCube object can be check by printing it. It will result the state of any cubie as seen in the outputs above.

2.2 Composition of operators

To act more than one transformation, it is possible to implement them as in (2.2) code. As said before, the order which the operators are applied by is fundamental, otherwise the final state could be a different one that the one one we expect. For instance, build a sequence of transformations

UBDLF, which means that the rotations (in order) of the front, left, down, back and up layers are applied on the Cube. Such operation can be implemented as follows:

Source Code 2.3: Composition of F and B

```python
1  from Rubik import RubiksCube, RubiksGroup
2
3  ## Allocate a RubiksCube object  ##
4  Cube = RubiksCube()
5  ## Allocate a RubiksGroup object ##
6  G = RubiksGroup()
7
8  ## Define the sequence of UBDLF layer rotations ##
9  operators = [G.U(), G.B(), G.D(), G.L(), G.F()]
10
11 ## Apply the operators one by one on the Cube object ##
12 for O in reversed(operators): O * Cube
```

First, the operators have been defined in a list, then they have been applied from right to left on Cube, which is the reason why the function reversed has been introduced on operators list in the for loop (line 9).
However, such operation may result awkward, especially when iterated many times. Another kind of operation has to be introduced, i.e. the composition between the generators of the Rubik's group. Such operation has been implemented by overwriting the @ (`__matmul__`) python operator, used by default for matrix multiplications. Taking back the example in (2.2) source code, we may rewrite it via the @ operator:

Source Code 2.4: Composition of F and B

```python
from Rubik import RubiksCube, RubiksGroup

## Implement the Cube ##
Cube = RubiksCube()
print("-"*20)
print(Cube)
print("-"*20)

## Implement the up and down rotations ##
F = RubiksGroup.F()
B = Rubiksgroup.B()

## Compose the operators ##
FB = F @ B

## Act on the Rubik's Cube ##
FB * Cube
print(Cube)
print("-"*20)
```

When printing the output, one may check that it will be the same as when applying F and B on the Cube object via * operator. To compose more operators, it is possible to deploy several times the @ operation, or rather a self method from RubiksGroup class:

Source Code 2.5: Composition of F and B

```python
from Rubik import RubiksCube, RubiksGroup

```

```python
## Define a RubiksCube object  ##
Cube = RubiksCube()
## Define a RubiksGroup object ##
G = RubiksGroup()

## Define the sequence of UBDLF layer rotations ##
operators = [G.U(), G.B(), G.D(), G.L(), G.F()]

## Compose the list of operators into a single one ##
UBDLF = G.compose_multipleOperators(operators)

## Apply the composed operators on the Cube object ##
UBDLF * Cube
```

The method in line 282 can be called for every python iterable (sets, tuples, lists and so on). As a generic rule, when applying a RubiksGroup transformation on a RubiksCube object (a generator or a composed element), we shall use the * operator, while when composing RubiksGroup elements, the @ operator has to be used.

3 Developer guide

In the following section, we analyse the structure of the program which allows to allocate the object and the operations described in the chapter before. To explain such implementation, some algebraic rules need to be highlighted.

3.1 Baseline

In baseline.py are defined all the basic functions and classes, the bricks on which the whole Cube is built on. All the elements are included in a single file, to distinguish from the classes which describe explicitly the Rubik's Cube and the Rubik's group.

3.1.1 Exponential class

The first object we meet is the class Exponential. The tuples of zeros, +1 and -1 shown in the outputs are nothing but an attribute of this class. The reason why such tuples are instantiated by the Exponential class lies in the fact that they represent the arguments of complex exponential functions:

$$e^{i\mathbf{x}\cdot\mathbf{k}} = e^{i(xk_x+yk_y+zk_z)} \longleftrightarrow (k_x, k_y, k_z) \tag{3.1}$$

Thus, in the first place, parameters (k_x, k_y, k_z) are needed to be passed in the constructor when allocating an Exponential object:

Source Code 3.1: Exponential class allocation

```python
from baseline import Exponential as Exp

## Allocating some Exponential elements ##
exp0 = Exp()                ## (0,0,0)  ##
exp1 = Exp(1)               ## (1,0,0)  ##
exp2 = T(1,-2)              ## (1,-2,0) ##
exp3 = Exp(x=1,y=2,z=3)     ## (1,2,3)  ##
exp4 = Exp(z=-1)            ## (0,0,-1) ##

## Print an Exponential object ##
print(Exp())
```

```
12 print("-"*10)
13
14 ## Check two Exponential objects to be equal ##
15 print(exp3==Exp(1,2,3))
```

Output:

```
(1, 2, 3)
----------
true
```

At line 4, we have passed default values to exp0 object, which are all initialized to be zero. At line 5, only the first element is non-null, while the others are set to be zero. Accordingly, the Exponential object at line 6 has only the first two parameters set to be non-zero. The exp3 object is initialized by three non-zero parameters, while the last one, exp4, has x and y null parameters, and k is set to be -1. At the flank of each Exponential allocated object, the correspondent tuple of values. To evaluate two Exponential objects to be equal, the `__eq__` method has been defined. We may take a look at the implementation of this class to clarify some ideas:

Source Code 3.2: Exponential class implementation

```
## Exponential class defines the position of the cubies ##
class Exponential:
    def __init__(self, x=0, y=0, z=0):
        self.x=x
        self.y=y
        self.z=z

     ## multiplicative operation between exponentials objects ##
    def __mul__(self, other):
        return Exponential(self.x+other.x, self.y+other.y, self.z+other.z)

    ## print method for exponentials ##
    def __repr__(self):
        return "(%s, %s, %s)" % (self.x, self.y, self.z)

    def __eq__(self, other):
        ## exponentials are equal when all instance attributes (i.e. self.x,self.y,self.z) match ##
        return all(e1 == e2 for e1, e2 in zip(self.__dict__.values(), other.__dict__.values()))
```

The `__init__` method takes three parameters, which are set to zero by default. The `__repr__` method allows to print the Exponential function, while the `__eq__` method to confront two Exponential object. `__mul__` allows operations between Exponential class objects.
The multiplicative operations reflect the composition of two complex exponential functions:

$$
\begin{aligned}
\exp\{i\,(xk_{1x}+yk_{1y}+zk_{1z})\} * \exp\{i(xk_{2x}+yk_{2y}+zk_{2z})\} &= \\
= \exp\{i(x(k_{1x}+k_{2x})+y(k_{1y}+k_{2y})+z(k_{1z}+k_{2z}))\} \longleftrightarrow (k_{1x},k_{1y},k_{1z}) * (k_{2x},k_{2y},k_{2z}) &= \\
= (k_{1x}+k_{2x},k_{1y}+k_{2y},k_{1z}+k_{2z})
\end{aligned}
\tag{3.2}
$$

3.1.2 Classes derived from Exponential

When defining a cubie, a tuple of parameters must be associated to, in order to measure the distance from their solved positions. Moreover, a vector to fix the orientation of the cubie is required, which has to be passed as additional argument. The parent class, which inherits in turn from Exponential, is the Cubie class, implemented as follows:

Source Code 3.3: Cubie class implementation

```
## Classes for cubies ##
class Cubie(Exponential):
    def __init__(self, vector=None, x=0, y=0, z=0):
        super().__init__(x, y, z)
        self.orientation = vector

    def __mul__(self, other):
        return self.__class__(self.x + other.x, self.y + other.y, self.z + other.z, self.orientation)

    def __repr__(self):
        return "(%s, %s, %s, %s)" % (self.x, self.y, self.z, self.orientation)

    def __eq__(self, other):
        ## Cubies are equal when all instance attributes (i.e. self.x,self.y,self.z and self.orientation) match ##
        return list(self.__dict__.values())[:3] == list(other.__dict__.values())[:3] and np.array_equal(self.orientation, other.orientation)
```

Parameters x,y,z are required as initializing arguments for a Cubie object, however, a vector is required too. Such vector aligns the starting orientation of the cubie. The `__mul__` operator permits to introduce the operation between Cubie and Exponential (and inherited classes) objects, with the constraint to preserve the orientation of the Cubie object. The `__repr__` gives a representation of the Cubie object, in the form of a tuple plus an array like the outputs in section (2.1). At last, the `__eq__` operation allows to compare the x,y,z parameters and the orientation vectors between two Cubie objects, to define them equal or not. From Cubie, in turn, Corner and Edge classes inherit all the methods:

Source Code 3.4: Corner and Edge implementation

```
## Define class for corner cubies ##
class Corner(Cubie):
    def __init__(self, x=0, y=0, z=0, vector=np.array([0.,1.,0.])):
        ## the vector param must be a corner state of orientation ##
        ## [0,1,0], [1,0,0], [0,0,1] numpy arrays are the three possible states ##
        if not tuple(vector) in list(p([0,1,0])):
            raise TypeError(f"Vector {vector} does not match any corner state of orientation")
        super().__init__(vector, x,y,z)


## Define class for edge cubies ##
class Edge(Cubie):
    def __init__(self, x=0, y=0, z=0, vector=np.array([1.,0.])):
        ## the vector param must be an edge state of orientation    ##
        ## [0,1], [1,0] numpy arrays are the three possible states ##
        if not tuple(vector) in list(p([1,0])):
            raise TypeError(f"Vector {vector} does not match any edge state of orientation")
        super().__init__(vector, x, y, z)
```

In the constructors, the orientation vector is initialized by a default value, which matches the state of the cubie in its solved configuration (as well as x,y,z are set to be zero). However, a generic vector is not allowed to be initialized, being only the permutations of the solved configuration feasible. Just for instance, a [0,1] vector is allowed for an Edge object, as far as it can be obtained permuting the [1,0] state of orientation. The same vector is not allowed to allocate a Corner object, because it cannot be obtained by permuting [0,1,0]. In such cases, an Error is raised when compiling.

3.1.3 Operator classes

In this section, a description on the classes which act on Corner and Edge ones is provided. The object from such classes may alter, for example, the state of position or orientation of a Cubie object, once some operations are well defined. The operators classes are two, Translation and Sigma:

Source Code 3.5: Translation and rotation implementation

```python
## Translation operators ##
class Translation(Exponential):
    def __init__(self,x=0,y=0,z=0):
        super().__init__(x,y,z)

    def __matmul__(self, cubie):
        return cubie.__class__(self.x+cubie.x, self.y+cubie.y, self.z+cubie.z, cubie.orientation)

## Rotation operators ##
class Sigma:
    def __init__(self, matrix):
        self.matrix = matrix

    def __mul__(self, cubie):
        ## return the same Cubie but with different orientation (@ stands for matmul operation) ##
        return cubie.__class__(cubie.x, cubie.y, cubie.z, self.matrix @ cubie.orientation )

    def __matmul__(self, other):
        ## composition between matrices ##
        return other.__class__(self.matrix@other.matrix)

    def __repr__(self):
        return str(self.matrix)

    ## Define @classmethod to build default constructors ##
    ## Flip matrix for edges ##
    @classmethod
    def X(cls):
        return cls(np.array([[0.,1.],[1.,0.]]) )

    ## Clockwise rotation matrix for corners ##
    @classmethod
    def C(cls):
        return cls(np.array([[0.,0.,1.], [1., 0.,0.], [0.,1.,0.]]))

    ## Anticlockwise rotation matrix for corners ##
    @classmethod
    def A(cls):
        return cls(np.array([[0.,1.,0.], [0., 0.,1.], [1.,0.,0.]]))
```

The Translation class inherits from Exponential to act on a Cubie object, no matter if Corner or Edge, as their positional properties are the same. The `__eq__` operation shifts the values of x,y,z class attributes, without afflicting the orientation of the cubie. Any Translation object can be in fact thought as nothing but an Exponential, equipped with an operation which allows to act on objects with an orientation attribute (while the Exponential class is not).
On the contrary, the Sigma class is endowed with an operation which leave unaffected the position of the cubies, but it transforms their orientation. Sigma does not inherit from any previous class,

thus the __mul__ and __matmul__ must be overwritten from the beginning. To allocate a Sigma object, a matrix must be furnished as an argument to the constructor. The __matmul__ method identifies any Sigma object with this own attribute. The * operator acts on a Cubie object, by a row by column multiplication on its orientation vector, the @ one is nothing but a composition of two matrix attributes from two Sigma objects, resulting thus in a new Sigma one. Some @classmethod decorators are introduced, to implement the canonic transformations on corners and edges when dealing with the Rubik's Cube. These transformations are the flip of the edges (given by X method) and the clockwise and anti-clockwise rotations of a corner cubie (C and A methods). For instance, when composing a C and a A matrices, or two X ones, identity function will be returned:

Source Code 3.6: Sigma class, examples of matmul operations

```python
from baseline import Sigma

A = Sigma.A()
C = Sigma.C()
X = Sigma.X()

print(A @ C)
print("-"*10)
print(X @ X)
```

Output:

```
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
----------
[[1. 0.]
 [0. 1.]]
```

i.e. the identity matrices in $\mathbb{R}_{3\times3}$ and $\mathbb{R}_{2\times2}$ spaces.

3.1.4 Permutations

Permutations class enable to permute numpy vectors by a given order of exchanges. Let's take a look at the constructor in baseline.py:

Source Code 3.7: Permutation constructor

```python
class Permutations:
    def __init__(self, cycle, cycle_2=None):
        ## introduce two-cycle notation ##
        self.cycle1 = cycle
        if cycle_2 is None: self.cycle2 = np.roll(cycle,-1)
        else: self.cycle2 = cycle_2
        if len(self.cycle1)!=len(self.cycle2):
            raise TypeError(f"Permuting lists {self.cycle1} and {self.cycle2}
            ↪ must have the same length: {len(self.cycle1)} is not
            ↪ {len(self.cycle2)}")
        for elem in self.cycle1:
            if elem % 1 != 0: raise TypeError(f"{elem} is not an integer")
            if elem not in self.cycle2: raise TypeError(f"{elem} not in
            ↪ {self.cycle2}")
```

There are two attributes which have to be initialized, i.e. self.cycle1 and self.cycle2. Such objects can be passed as one or two lists of integers in the constructor (only one argument is required). When just one list is passed, self.cycle2 attribute will be equal to self.cycle1, except for taking its first element and stocking it at the end. Some constraints are expressed by raising errors: the two

lists must have the same length and the same elements, as well as any number must be an integer. However, as seen at line 13, floats are admitted, but must have a null decimal part.
Other features in Permutations class are the overwritten operators:

Source Code 3.8: Permutation operators

```python
    ## verify two permutations to be equivalent ##
    def __eq__(self, other):
        return np.array_equal(self.cycle1, other.cycle1) and np.array_equal(self.cycle2, other.cycle2)

    ## action on a vector ##
    def __mul__(self, vector):
        vector[self.cycle1] = vector[self.cycle2]
        return vector

    def __repr__(self):
        return str(self.cycle1) + '\n' + str(self.cycle2)
```

Two Permutations objects are equal when both of their cycles, as set by the `__eq__` operator. To represent a Permutations object, the two cycles are printed by `__repr__` function in column. When applied to a numpy vector, a Permutations object shifts the vectorial elements from the indices in self.cycle1 to the indices in self.cycle2 positions. Let's see a direct example:

Source Code 3.9: Permutation example

```python
from baseline import Permutations as Perm
import numpy as np

P = Perm([1,2,3])

print("P:")
print(P)
print("-"*10)

## allocate a (0,1,2,3,4) numpy vector ##
v = np.arange(5)
print(v)
print("-"*10)
## permute v elements ##
print(P * v)
```

Output:

```
P:
[1, 2, 3]
[2, 3, 1]
----------
[0 1 2 3 4]
----------
[0 2 3 1 4]
```

The first element on the vector is shifted at the second one, the second in the third one and the third one back to the first. The output representation of P reflects the two-cycle notation to describe a $P_{(a,b,c,d)}$ permutation:

$$P = \begin{pmatrix} a & b & c & d \\ b & c & d & a \end{pmatrix} \tag{3.3}$$

When applying a P permutation on a v vector, the v_a element in the a-th position is shifted in the b-th one, v_b, and so on. Such formalism can be also represented in a functional fashion:

$$P : (a, b, c, d) \mapsto (b, d, c, a) \tag{3.4}$$

By this notation, $P(a) = b$, $P(b) = c$ and so on. A python vocabulary fits very well this purpose, and such feature is provided by the self.convert function:

Source Code 3.10: Permutation convert() method

```
    def convert(self):
        ## first, decompose the permutation into single exchanges ##
        ## such a purpose is given by splitting the two cycles in a nx2 matrix,
        ↪  each row being an exchange ##
        decoupling = np.array([self.cycle1, self.cycle2]).transpose()
        ## now return a dictionary ##
        return {x[0]: x[1] for x in decoupling}
```

In code (3.10), the P matrix from eq. (3.3) is morphed into the functional form in eq. (3.4). This method will be useful when composing two different permutations:

$$\begin{pmatrix} a & b & c \\ b & c & a \end{pmatrix} \circ \begin{pmatrix} a & b & c & d \\ c & a & d & b \end{pmatrix} = \begin{pmatrix} b & d \\ d & b \end{pmatrix} \tag{3.5}$$

In the above equation, the b element in the first permutation is mapped into c, then, in the second permutation, c is mapped into d, i.e. $b \mapsto c \mapsto d$. It follows that the resulting composition will be $b \mapsto d$, and so for $d \mapsto b$, while $a \mapsto a$ and $c \mapsto c$, thus they are deleted. The functional formalism, implemented via convert() self method, turns to be very useful, and is implemented in the `__matmul__` operation of composition:

Source Code 3.11: Permutation __matmul__ operator

```
167     def __matmul__(self, other):
168         ##################################################
169         ##    convert the exchanges into a dictionary    ##
170         ## use the self.convert() function defined above ##
171         ##################################################
172         dic1 = self.convert()
173         dic2 = other.convert()
174         for key, value in dic1.items():
175             try: dic1[key] = dic2[value]
176             except: continue
177         for key, value in dic2.items():
178             if key not in dic1: dic1[key] = value
179         for key in dic1.copy():
180             if key == dic1[key]: dic1.pop(key)
181         A = np.array(list(dic1.items())).transpose()
182         try:
183             return Permutations(A[0],A[1])
184         ## when A == [] no elements are permuted              ##
185         ## in such case, return a [0] list                    ##
186         ## i.e. and identity permutation on the first element ##
187         except:
188             return Permutations([0])
```

The code above follows the scheme from eq. (3.5). When a null permutation is to be returned, an error raises when applying it on a vector. To compensate such situation, it is returned a Permutations object on a [0] list (line 22). As far as any vector to permute would have at least one element, a Permutations([0]) object permutes the first element from an array with itself, returning in fact an identity operator $P : 0 \mapsto 0$. When composing such permutation with other ones, the result will give nothing but the second permutation itself. The goal to implement the identity element, from the math group of permutations, is achieved.

3.2 Rubik's classes

In the last section, we discuss about the two classes which defines the rules of the game for the Rubik's Cube. The first one returns an object which describes the Cube itself, while the second one focuses on the properties of the Rubik's group, and the algebraic rules when its elements are acting on the Rubik's Cube.

3.2.1 RubiksCube class

Any RubiksCube object consists of two attributes, the first one an array allocating the solved state of the Cube, the other one the actual state. It follows the implementation of this class with its instances and methods:

Source Code 3.12: RubiksCube class

```
31 class RubiksCube:
32     def __init__(self, state_vector=None):
33         ## define the solved state of the Cube ##
34         self.solved = np.concatenate((np.array([Edge() for _ in range(12)]),
           ↪  np.array([Corner() for _ in range(8)])),
35                                       axis=0)
36         ## allocate the actual state of the Cube ##
37         self.Cube = state_vector if state_vector is not None else
           ↪  copy(self.solved)
38 
39     ## print function should return the self.Cube, i.e. the state vector ##
40     def __repr__(self):
41         return str(self.Cube).replace(') ', ')\n ').replace(' (', '(')[1:-1]
42 
43     def __eq__(self, other):
44         return all(e1 == e2 for e1, e2 in zip(self.Cube, other.Cube))
45 
46     ## reset the Cube to its solved state ##
47     def reset(self):
48         self.Cube = copy(self.solved)
49 
50     ## function to determine whether the Cube is solved or not ##
51     def is_solved(self):
52         return all(self.Cube == self.solved)
```

In the constructor, a numpy vector can be passed to the class, in order to implement the initial state of the Cube. If no vector is passed, the solved state (defined in line 4-5) is passed by default. The self.solved attribute is always passed per copy to the self.Cube state, in order not to mingle the memory addresses of the two pointers.
The representation of a RubiksCube object consists of printing all the cubies per row, from which results an output like from (2.1) and (2.2) lines of code. The __eq__ operator confronts cubie by cubie two RubiksCube objects to be equal, which is made possible by providing a __eq__ operator by the Cubie class itself. The reset method turns the self.Cube attribute (i.e. the state vector of the Cube) to match the solved configuration. At last, the is_solved(self) method checks the self.Cube to be in the solved state or not. To see how to encode this class, watch back the example codes in section (2.1).

3.2.2 RubiksGroup class

The RubiksGroup class allows to implement the algebraic transformations on the Rubik's Cube and the composition between elements belonging to this group. In the first place, a generator G from Rubik's group can be decomposed in its action on the edges and the corners as follows:

$$\hat{G}_e = \hat{P}_{\sigma(a,b,c,d)}\hat{O}(a,b,c,d), \quad \hat{G}_c = \hat{P}_{\sigma(f,g,h,l)}\hat{O}(f,g,h,l) \tag{3.6}$$

a, b, c, d are the edges involved in the transformation, f, g, h, l the corners. The hats over G, P and O mean that we are dealing with algebraic operators. $\hat{P}_{\sigma(a,b,c,d)}$ stands for the permutation on a, b, c, d, and so for the corners. $\hat{O}$ operator groups all the translations and sigma matrices, to change orientation and position for the involved cubies. All translations and sigma matrices act on a single cubie, thus commute each other with, and can be placed into a wider $\hat{O}$ operator. For instance, the front (F) transformation can be written as

$$\hat{F}_e = \hat{P}^e_{\sigma(a,b,c,d)} e^{i\frac{2\pi}{l}(y_a+x_a+y_b-x_b-x_c-y_c-y_d+x_d)} \hat{\sigma}^a_x \hat{\sigma}^b_x \hat{\sigma}^c_x \hat{\sigma}^d_x \tag{3.7}$$

$$\hat{F}_c = \hat{P}^c_{\sigma(f,g,h,l)} e^{i\frac{2\pi}{l}(y_f-x_g-y_h+x_l)} \hat{\sigma}^f_{x_A} \hat{\sigma}^g_{x_C} \hat{\sigma}^h_{x_A} \hat{\sigma}^l_{x_C} \tag{3.8}$$

Thus, the $\hat{O}$ operators will be

$$\hat{O}_e = e^{i\frac{2\pi}{l}(y_a+x_a+y_b-x_b-x_c-y_c-y_d+x_d)} \hat{\sigma}^a_x \hat{\sigma}^b_x \hat{\sigma}^c_x \hat{\sigma}^d_x \tag{3.9}$$

$$\hat{O}_c = e^{i\frac{2\pi}{l}(y_f-x_g-y_h+x_l)} \hat{\sigma}^f_{x_A} \hat{\sigma}^g_{x_C} \hat{\sigma}^h_{x_A} \hat{\sigma}^l_{x_C} \tag{3.10}$$

i.e. the $\hat{O}$ operators are nothing but the whole transformation, except the permutations at the beginning. Just to remember, the exponential functions are encoded into the Translation class, while the σ matrices into the Sigma class. The notation adopted in the equations from (3.6) to (3.10) is incomplete: an identity operator is applied on all the cubies which are not involved in the transformation. However, when programming such transformations, it would be highly expensive, in computational time, to act with identity operators on 12 cubies, the other 8 being transformed. Thus, for every transformation to implement, it is required to label the cubies involved (they are numerated from 0 to 19), the Translation, Sigma and Permutations operators acting on them. The numerical labels of the cubies are shown in fig. (3.1). All these parameters are passed in the constructor:

Source Code 3.13: RubiksGroup constructor

```
class RubiksGroup():
    def __init__(self, *args):
        ## EDGES ##
        ## The indices of the edges are given by args[0]       ##
        ## args[0] is the first list passed in the constructor ##
        ## translations: map from indices to exp operators ##
        self.edge_transl = {key : value for key, value in zip(args[0], args[1])}
        ## flips: map from indices to sigma_x matrices ##
        self.edge_flip = {key : value for key, value in zip(args[0], args[2])}
        ## edge permutations ##
        self.Pe = args[3]

        ## CORNERS ##
        ## The indices of the corners are given by args[4]     ##
        ## args[0] is the fifth list passed in the constructor ##
        ## translations: map from indices to exp operators ##
        self.corner_transl = {key : value for key, value in zip(args[4],
        ↪  args[5])}
        ## rotations: map from indices to sigma matrices ##
        self.corner_rot = {key : value for key, value in zip(args[4], args[6])}
        ## corner permutations ##
        self.Pc = args[7]
```

The first four args are, respectively, the edge involved in the transformation, the Translation and Sigma operators to be applied and the Permutations, the last four are the same parameters, but for the corners. Translations and rotations of the cubies are defined in a dictionary, which takes the labelling number of the cubie as key and returns the operator to act with. The action on a RubiksCube object is defined in the `__mul__` operator:

Source Code 3.14: RubiksGroup `__mul__` operator

```
    def __mul__(self, Cube):
        ## single cubie transformations ##
        for te in self.edge_transl.items():
```

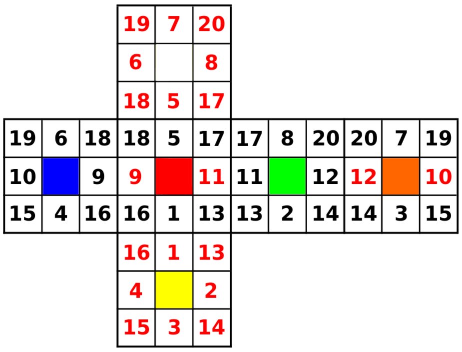

Figure 3.1: The Rubik's Cube, in the solved configuration, opened to show all of its faces. The cubies are enumerated in order to describe the transformations of the Rubik's group. The red numbers show the oriented faces of the cubies, i.e. the faces correctly oriented in the solved configuration of the Cube. The colours at the center of each face are those of the Cube: red stands for the front face F, blue for the left F, green for the right R, orange for the back B, yellow for the bottom U, white for the top D.

```
94              Cube.Cube[te[0]] *= te[1]
95          for fe in self.edge_flip.items():
96              Cube.Cube[fe[0]] = fe[1] * Cube.Cube[fe[0]]
97          for tc in self.corner_transl.items():
98              Cube.Cube[tc[0]] *= tc[1]
99              #print(Cube.Cube[tc[0]])
100         for rc in self.corner_rot.items():
101             Cube.Cube[rc[0]] = rc[1] * Cube.Cube[rc[0]]
102         ## permutation of the cubies ##
103         Cube.Cube = self.Pe * Cube.Cube
104         Cube.Cube = self.Pc * Cube.Cube
105         return Cube
```

The code above follows the scheme in eq. (3.6), at first the operations on the single cubies (translations, sigma rotations) are applied, then the permutations. The dictionary attributes of RubiksGroup class contain the cubie labels as key, and the single cubie operators as value. From line 3 to line 11 such dictionaries are applied on the Cube.Cube object, which is nothing but the vector state of the RubiksCube class. Afterwards, the cubies are permuted.
The next step is to compose two different generators, and consequently any composition of generators with another one. To do so, first watch how to compose two elements from the Rubik's group, call them G_1 and G_2, in the unitary formalism:

$$\hat{G}_1 \circ \hat{G}_2 = \hat{P}_{\sigma(a_1,b_1,c_1,d_1)}\hat{O}(a_1,b_1,c_1,d_1) \circ \hat{P}_{\sigma(a_2,b_2,c_2,d_2)}\hat{O}(a_2,b_2,c_2,d_2) \tag{3.11}$$

To simplify the notation, instead of calling a_1, b_1 and so on, put a 1 or 2 subscript on the operators. Then, an identity operator, $\hat{P}_2\hat{P}_2^\dagger$, can be introduced after the first permutation:

$$\hat{G}_{tot} = \hat{G}_1 \circ \hat{G}_2 = \hat{P}_1\hat{P}_2\hat{P}_2^\dagger\hat{O}_1\hat{P}_2\hat{O}_2 = \hat{P}_1\hat{P}_2\hat{O}_1'\hat{O}_2 = \hat{P}_{tot}\hat{O}_{tot} \tag{3.12}$$

meaning, by $\hat{O}_1'$, that the operator $\hat{O}(a_1,b_1,c_1,d_1)$ has been modified by the action of the $\hat{P}_2$ permutations:

$$\hat{O}_1' = \hat{P}_2^\dagger\hat{O}_1\hat{P}_2 \tag{3.13}$$

By this transformation, the a_1, b_1 and so on elements are mapped by the $\hat{P}_{\sigma(a_2,b_2,c_2,d_2)}$ permutation. For instance, take $b_2 = b_1$ and $c_2 = d_1$, while a_2 and d_2 being different elements. In such example, the $\hat{P}_2$ permutation can be written as

$$\hat{P}_2 = \begin{pmatrix} a_2 & b_1 & d_1 & d_2 \\ b_1 & d_1 & d_2 & a_2 \end{pmatrix} \tag{3.14}$$

Thus $b_1 \mapsto d_1$, and $d_1 \mapsto d_2$, resulting in

$$\hat{O}'_1 = O_1(a_1, d_1, c_1, d_2) \tag{3.15}$$

To implement a composition of elements from Rubik's group, is thus required:

1. to switch the labels of the first operator to compose into the permuted ones, i.e. to execute the operation $\hat{O}_1 \mapsto \hat{O}'_1$ as in eq. (3.15). In a computational perspective, all the keys from the dictionary attributes must be changed and updated following such rule;
2. to compose $\hat{O}'_1$ and $\hat{O}_2$ among them in a $\hat{O}_{tot}$ operator, as in eq. (3.12);
3. to compose $\hat{P}_1$ and $\hat{P}_2$ among them in a $\hat{P}_{tot}$ permutation, as in eq. (3.12);

The same agenda must be accomplished for both corner and edges. The first point is achieved in two separate methods, namely compose_translations and compose_orientations. We can analyze the first one, the second one being the same but with the orientations as values of the dictionaries rather than the translations. The code is the following:

Source Code 3.15: RubiksGroup compose_translation() method

```
112     ## compose translations ##
113     def compose_translations(self, dic1, dic2, permutation):
114         ## newDic will be the dictionary of the composed operator ##
115         newDic = {}
116         ## permutations of the second operator must be applied on the first one ##
117         ## the reason of such operation depends from basics of group theory ##
118         for key in [*dic1]:
119             if key in permutation.cycle1: newDic[permutation.convert()[key]] = dic1[key]
120         ## find common keys between dic2 and newDic ##
121         common_elements = set(newDic.keys()) & set(dic2.keys())
122         ## compose common elements in newDic ##
123         for elem in common_elements: newDic[elem] *= dic2[elem]
124         ## add the items from dic1 and dic2 whose keys are not in newDic ##
125         newDic = dict(list(dic2.items()) + list(newDic.items()))
126         newDic = dict(list(dic1.items()) + list(newDic.items()))
127         ## return keys and values from newDic ##
128         return [*newDic], [*newDic.values()]
```

The switch of the labels (first point of the agenda) is achieved at line 7. In the remaining lines, the features from the new $\hat{O}'_1$ operator (corner or edge translations, rather than their sigma operators) are composed with the $\hat{O}_2$ ones, by combining the two dictionaries together.
The whole operation is assembled by the `__matmul__` operator:

Source Code 3.16: RubiksGroup `__matmul__` operator

```
141     ## Composition ##
142     def __matmul__(self, other):
143         ## COMPOSE PERMUTATIONS ##
144         perm_e = self.Pe @ other.Pe
145         perm_c = self.Pc @ other.Pc
146         ## COMPOSE OPERATORS ##
147         ## edge translations ##
148         edges, edge_translations = self.compose_translations(self.edge_transl, other.edge_transl, other.Pe)
149         ## corner translations ##
150         corners, corner_translations = self.compose_translations(self.corner_transl, other.corner_transl, other.Pc)
151         ## edge orientation ##
```

```python
152         edge_orientations = self.compose_orientations(self.edge_flip, other.edge_flip, other.Pe)
153         ## corner orientation ##
154         corner_orientations = self.compose_orientations(self.corner_rot, other.corner_rot, other.Pc)
155 
156         return RubiksGroup(edges, edge_translations, edge_orientations, perm_e, corners, corner_translations, corner_orientations, perm_c)
```

At lines 3-4, the permutations are combined, while corner and edge translations and sigma operators are combined in lines 7-21. A RubiksGroup object is at last returned at line 23. The same operation can be executed for multiple operators by calling the compose_multipleOperators method. A list of RubiksGroup objects is passed as an argument, and thus they are composed by the order from the list:

Source Code 3.17: RubiksGroup compose_multipleOperators() method

```python
    ## From a list of operators, this method returns their composition ##
    ## The argument can be any iterable (lists, tuples, sets etc...)   ##
    @classmethod
    def compose_multipleOperators(cls, operators_to_compose):
        return reduce(lambda x, y: x @ y, operators_to_compose)
```

4 Requirements

The software is supported by python 3.9, a list for the required libraries to install is provided:

- numpy 1.21.2
- unittest

5 Disclaimer

The Authors decline any responsibility connected to the usage of this software.

6 Acknowledgments

S.C. would like to thank his former colleague Paolo Gibertini, who has been a mentor and a tutor in the art of programming, passing much passion and skills. The Authors would also like to thank Lorenzo Moro, who actively participated to the development of the Reinforcement learning agent, exploiting this environment in paper.