

# T.I.P.E: Rubik's Cube

## 2x2x2

### I) Définitions des constantes et opérations de bases.

Les constantes sont en principe toutes écrites en majuscule, à l'exception des plus usitées qui sont alors toutes en minuscule.

\*On charge tout d'abord le module utilisé:  
*with(plots) :*

#### A) Les couleurs

\*Tableau (C) des 6 couleurs utilisées dans le cube (version officielle):

Ce tableau sert essentiellement à obtenir les couleurs de chaque face pour les options des plots, la manipulation pour les mouvements du Rubik's Cube se faisant alors au niveau des indices, on a donc Chiffre⇒Couleur(en Maple)

```
C[1] := red:  
C[2] := "DarkGreen":  
C[3] := "OrangeRed":  
C[4] := blue:  
C[5] := yellow:  
C[6] := white:
```

\*Attributions des couleurs à chaque indice de C (initiales anglaises pour éviter les confusions bleu/blanc):

Ces notations permettent exclusivement de faciliter l'entrée des couleurs de chaque face d'un Rubik's Cube donné, en pratique on s'en sert de la manière suivante: Couleur(en "francais")⇒Chiffre⇒Indice⇒Couleur(en Maple)

```
r := 1 :  
g := 2 :  
o := 3 :  
b := 4 :  
y := 5 :  
w := 6 :
```

## B) Le Rubik's Cube

Le Rubik's Cube est un tableau composé par chacun de ses 8 petits cubes.

Les petits cubes sont eux même des tableaux composés de leurs coordonnées et des couleurs de leurs 3 faces.

Les coordonnées sont eux aussi des tableaux donnés par 3 chiffres (d'après les notations adoptées)

Les couleurs sont finalement toujours des tableaux donnés par les 3 couleurs des faces du petit cube concerné.

### \*Définitions des 8 cubes d'un Rubik's Cube fini:

Le numéro de chaque cube n'a aucune importance (en revanche, cela est capital lors de l'affichage 3D, d'où la fonction Rafraîchir), ils sont pris dans l'ordre de la notation choisie.

$$CUBE.FINI[1] := \begin{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} C[w] \\ C[r] \\ C[b] \end{bmatrix} \end{bmatrix};$$
$$CUBE.FINI[2] := \begin{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} C[g] \\ C[r] \\ C[w] \end{bmatrix} \end{bmatrix};$$

$$CUBE.FINI[3] := \begin{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}, \begin{bmatrix} C[b] \\ C[r] \\ C[y] \end{bmatrix} \end{bmatrix};$$

$$CUBE.FINI[4] := \begin{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} C[y] \\ C[r] \\ C[g] \end{bmatrix} \end{bmatrix};$$

$$CUBE.FINI[5] := \begin{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} C[b] \\ C[o] \\ C[w] \end{bmatrix} \end{bmatrix};$$

$$CUBE.FINI[6] := \begin{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} C[w] \\ C[o] \\ C[g] \end{bmatrix} \end{bmatrix};$$

$$CUBE.FINI[7] := \begin{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} C[y] \\ C[o] \\ C[b] \end{bmatrix} \end{bmatrix};$$

$$CUBE.FINI[8] := \left[ \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, \begin{bmatrix} C[g] \\ C[o] \\ C[y] \end{bmatrix} \right]:$$

**\*Notations du premier tableau d'un cube (les coordonnées) et du second (les couleurs):**

Dans le but de clarifier les procédures.

*coor := 1 ;  
coul := 2 ;*

**\*Définitions du Rubik's Cube fini:**

*RUBIX.FINI := [CUBE.FINI[1], CUBE.FINI[2], CUBE.FINI[3], CUBE.FINI[4], CUBE.FINI[5], CUBE.FINI[6], CUBE.FINI[7], CUBE.FINI[8]] :*

Compte tenu de l'utilisation ultra fréquente de ce dernier dans les arguments des autres fonctions, on prendra la notation *rubix := RUBIX.FINI* : On conserve les deux notations au cas où la variable "rubix" serait accidentellement remplacée.

**\*Fonction de correspondance n° Petit Cube  $\Rightarrow$  Coordonnée:**

C'est une fonction qui donne les coordonnées correspondantes à un petit cube désigné par son numéro dans la notation choisie.

*NumeroVersCoordonnees :=proc (num,\$)*

*local x, y, z;*

*#La première coordonnée se récupère très facilement*

*x := 0<sup>num mod 2</sup>;*

*#Les deux autres nécessitent une boucle conditionnelle*

```
if num = 1 then y := 0; z := 1;
elif num = 2 then y := 0; z := 0;
elif num = 3 then y := 1; z := 1;
elif num = 4 then y := 1; z := 0;
elif num = 5 then y := 0; z := 0;
elif num = 6 then y := 0; z := 1;
elif num = 7 then y := 1; z := 0;
elif num = 8 then y := 1; z := 1;
fi;
```

*#On renvoie finalement le résultat*

*[x, y, z];*

*end proc:*

**\*Fonction de correspondance Coordonnée  $\Rightarrow$  n° Petit Cube:**

C'est une fonction qui donne le numéro d'un petit cube dans la notation choisie en fonction de ses coordonnées .

*CoordonneesVersNumero :=proc (x, y, z,\$)*

*#Une boucle conditionnelle est la façon la plus claire pour avoir ce qu'on veut*

```
if (x, y, z) = (0, 0, 1) then 1
elif (x, y, z) = (1, 0, 0) then 2
elif (x, y, z) = (0, 1, 1) then 3
elif (x, y, z) = (1, 1, 0) then 4
```

```

        elif (x, y, z) = (0, 0, 0) then 5
        elif (x, y, z) = (1, 0, 1) then 6
        elif (x, y, z) = (0, 1, 0) then 7
        elif (x, y, z) = (1, 1, 1) then 8
    fi
end proc;

```

#### **\*Fonction de correspondance Couleurs $\Rightarrow$ n° Petit Cube:**

C'est une fonction qui donne le numéro d'un petit cube dans la notation choisie et dans la position de référence (i.e. face rouge en face, face jaune en haut) en fonction de ses coordonnées . Cette convention de position de référence pose d'ailleurs de gros problèmes d'unicité de l'orientation, faisant de la fonction EstCycle une fonction assez hasardeuse, ne fonctionnant que dans un cadre bien précis.

*CouleursVersNumero :=proc (c1, c2, c3,\$)*

#Une boucle conditionnelle est la façon la plus claire d'avoir ce qu'on veut

```

        if {c1, c2, c3} = {C[w], C[r], C[b]} then 1
        elif {c1, c2, c3} = {C[g], C[r], C[w]} then 2
        elif {c1, c2, c3} = {C[b], C[r], C[y]} then 3
        elif {c1, c2, c3} = {C[y], C[r], C[g]} then 4
        elif {c1, c2, c3} = {C[b], C[o], C[w]} then 5
        elif {c1, c2, c3} = {C[w], C[o], C[g]} then 6
        elif {c1, c2, c3} = {C[y], C[o], C[b]} then 7
        elif {c1, c2, c3} = {C[g], C[o], C[y]} then 8
    end if;

```

end proc;

#### **\*Fonction de correspondance Couleurs $\Leftarrow$ n° Petit Cube:**

Fonction qui fait correspondre à une position (numéro d'un petit cube) un triplet de couleurs correspondant à celui du petit cube visé dans la notation choisie et dans la position de référence.

*NumerоВersCouleurs :=proc (a,\$)*

#Une boucle conditionnelle est la façon la plus claire d'avoir ce qu'on veut

```

        if a = 1 then C[w], C[r], C[b];
        elif a = 2 then C[g], C[r], C[w];
        elif a = 3 then C[b], C[r], C[y];
        elif a = 5 then C[y], C[r], C[g];
        elif a = 5 then C[b], C[o], C[w];
        elif a = 6 then C[w], C[o], C[g];
        elif a = 7 then C[y], C[o], C[b];
        elif a = 8 then C[g], C[o], C[y];
    end if;

```

end proc;

#### **\*Fonction de rafraîchissement du Rubik's cube, qui le place dans sa notation standard:**

Cette fonction est strictement nécessaire pour afficher correctement le cube en 3D. Elle réordonne les petits cubes du rubix pour les placer correctement les uns par rapport aux autres. En effet, DessineVolume considère que Rubix[1] correspond bien au petit cube 1, etc...

*Rafraichir :=proc(Rubix1,\$)*

local k, Rubix2;

#On fait correspondre à chaque coordonnée son numéro correspondant

```

for k to 8 do
    Rubix2[CoordonneesVersNumero(Rubix1[k][coor][1], Rubix1[k][coor][2],
    Rubix1[k][coor][3])] := Rubix1[k]
end do;
#On renvoie le nouveau Rubik's Cube
    Rubix2;
end proc:

```

## C) Les états du Rubik's Cube

### ***a)** La finitude*

#### **\*Fonction testant si le Rubik's Cube considéré est fini:**

On vérifie simplement que les 4 petites faces de chacune des 6 grandes faces sont de la même couleur. Ainsi la finitude ne dépend pas de l'orientation.

```

EstFini :=proc(Rubix2,$)
    local k, x, y, z, Rubix;
    #On pense bien à rafraîchir le Rubik's cube pour faire correspondre ses petits cubes
        Rubix := Rafraîchir(Rubix2);
    #Un booléen donne alors directement la réponse
        Rubix[1][coul][2] = Rubix[2][coul][2] = Rubix[3][coul][2] = Rubix[4][coul][2]
    and Rubix[5][coul][2] = Rubix[6][coul][2] = Rubix[7][coul][2] = Rubix[8][coul][2]
    and Rubix[2][coul][1] = Rubix[4][coul][3] = Rubix[6][coul][3] = Rubix[8][coul][1]
    and Rubix[1][coul][3] = Rubix[3][coul][1] = Rubix[5][coul][1] = Rubix[7][coul][3]
    and Rubix[1][coul][1] = Rubix[2][coul][3] = Rubix[5][coul][3]
        = Rubix[6][coul][1]
    and Rubix[3][coul][3] = Rubix[4][coul][1] = Rubix[7][coul][1] = Rubix[8][coul][3]
end proc:

```

### ***b)** La faisabilité*

Comme il existe une suite de mouvements permettant d'interchanger 2 petits cubes quelconques sans changer leur orientation, on ne considère que les orientations des petits cubes.

Il est alors nécessaire et suffisant d'avoir:

- $\forall \text{Cube1}, \text{Cube2} \in \text{Rubix}, \{\text{Cube1}[coul]\} \neq \{\text{Cube2}[coul]\}$
- $\forall \text{Cube} \in \text{Rubix}, \exists \text{CUBE.FINI} \in \text{RUBIX.FINI} \mid \exists n \in \mathbb{N} \mid \text{Cube}[coul] = c^n(\text{CUBE.FINI}[coul])$  : où  $c(E)$  est un cycle de l'ensemble  $E$
- $\sum n \equiv 0 \pmod{3}$

#### **\*Fonction qui vérifie l'axiome 1:**

Tous les petits cubes d'un Rubik's Cube faisable doivent nécessairement avoir leur triplet de couleurs différents entre eux.

*Axiome1 :=proc(Rubix,\$)*

*#On vérifie qu'il y ait bien 8 triplets différents de couleurs*  
*evalb(nops( {*

```

        {Rubix[1][coul][1], Rubix[1][coul][2], Rubix[1][coul][3]},
        {Rubix[2][coul][1], Rubix[2][coul][2], Rubix[2][coul][3]},
        {Rubix[3][coul][1], Rubix[3][coul][2], Rubix[3][coul][3]},
        {Rubix[4][coul][1], Rubix[4][coul][2], Rubix[4][coul][3]},

```

```

    {Rubix[ 5 ][coul][ 1 ], Rubix[ 5 ][coul][ 2 ], Rubix[ 5 ][coul][ 3 ]},
    {Rubix[ 6 ][coul][ 1 ], Rubix[ 6 ][coul][ 2 ], Rubix[ 6 ][coul][ 3 ]},
    {Rubix[ 7 ][coul][ 1 ], Rubix[ 7 ][coul][ 2 ], Rubix[ 7 ][coul][ 3 ]},
    {Rubix[ 8 ][coul][ 1 ], Rubix[ 8 ][coul][ 2 ], Rubix[ 8 ][coul][ 3 ]},
) ) = 8 );
end proc;

```

#### \*Fonction qui vérifie si un triplet de couleur est cycle du rubix fini:

Elle renvoie  $\infty$  si le triplet n'appartient à aucun cycle. Le gros problème de cette fonction, c'est que la valeur du cycle n'est même pas unique..Donc en absolu elle ne peut pas fonctionner mais elle fait correctement ce pour quoi on s'en sert, i.e. vérifier l'existence d'un cycle et établir une bijection entre  $\{0,1,2\}$  et les trois cas d'orientation possibles.

1ère version:

```

EstCycle :=proc(Couleur,$)
local k, n;
n := infinity;
#On vérifie que le vecteur de couleur donné est cycle du rubix fini
for k to 8 do
  if Couleur[1] = (RUBIX.FINI)[k][coul][1] and Couleur[2] = (RUBIX
.FINI)[k][coul][2] and Couleur[3] = (RUBIX.FINI)[k][coul][3] then n := 0;
break;
  elif Couleur[1] = (RUBIX.FINI)[k][coul][2] and Couleur[2] = (RUBIX
.FINI)[k][coul][3] and Couleur[3] = (RUBIX.FINI)[k][coul][1] then n := 1;
break;
  elif Couleur[1] = (RUBIX.FINI)[k][coul][3] and Couleur[2] = (RUBIX
.FINI)[k][coul][1] and Couleur[3] = (RUBIX.FINI)[k][coul][2] then n := 2;
break;
  end if;
end do;
#On renvoie l'indice du cycle
n;
end proc;

```

Un deuxième version tente de fixer une certaine unicité:

```

EstCycleII :=proc(Couleur, a,$)
local n, Rubix1, Rubix2;
n := infinity;
Rubix1 := Rafraichir(BougerRubix(RUBIX.FINI,
ChangerDeCoin4(OuEstPetitCube(RUBIX.FINI, (convert(Couleur, list))[])),
volume=false, patron=false, cube=true));
Rubix2 := Rafraichir(RUBIX.FINI);
Rubix2[a][coul] := Couleur;
Rubix2 := Rafraichir(BougerRubix(Rubix2, ChangerDeCoin4(a), volume
=false, patron=false, cube=true));
#On vérifie que le vecteur de couleur donné est cycle du rubix fini
if Rubix1[4][coul][1] = Rubix2[4][coul][1] and Rubix1[4][coul][2]
= Rubix2[4][coul][2] and Rubix1[4][coul][3] = Rubix2[4][coul][3] then n := 2;
elif Rubix1[4][coul][1] = Rubix2[4][coul][2] and Rubix1[4][coul][2]
= Rubix2[4][coul][3] and Rubix1[4][coul][3] = Rubix2[4][coul][1] then n := 1;
elif Rubix1[4][coul][1] = Rubix2[4][coul][3] and Rubix1[4][coul][2]
= Rubix2[4][coul][1] and Rubix1[4][coul][3] = Rubix2[4][coul][2] then n := 0;
end if;

```

#On renvoie l'indice du cycle

*n;*

**end proc:**

**\*Fonction qui vérifie l'axiome 2 et 3:**

L'orientation de chaque petit cube d'un Rubik's Cube faisable doit nécessairement être un cycle d'un des petits cubes du cube fini.

*Axiome2et3 :=proc(Rubix,\$)*

**local** *n, k;*

*n := 0;*

#On récupère l'indice des cycle de chaque cube

**for** *k* **to** 8 **do**

*n := n + EstCycle(Rubix[k][coul]);*

**end do;**

#On vérifie d'un coup les deux axiomes. En effet  $\infty \bmod 3 = \infty$  pour Maple

*evalb(n mod 3 = 0);*

**end proc:**

**\*Fonction qui vérifie si un cube est faisable:**

Vérifie juste les trois axiomes.

*EstFaisable :=proc(Rubix,\$)*

*(Axiome1(Rubix) and Axiome2et3(Rubix))*

**end proc:**

## II) Les 12 mouvements possibles

On programme successivement tous les mouvements possibles. Chacun change un à un les coordonnées et les couleurs des petits cubes afin de les faire correspondre avec ce qu'ils devraient être après le mouvement considéré.

### A) Selon la face gauche

**\*Fonction qui change les données d'un petit cube ayant subi une rotation du bloc L(left)  
(dans le sens direct):**

*RotationCubeL := proc(Cube,\$)*

**local** *x, y, z, C, a, b, c, d, e, f;*

#On récupère les coordonnées du petit cube donné

*x := Cube[coor][1];*

*y := Cube[coor][2];*

*z := Cube[coor][3];*

#On récupère l'orientation des couleurs du petit cube donné

*C[0] := Cube[coul][1];*

*C[1] := Cube[coul][2];*

*C[2] := Cube[coul][3];*

#On détermine les nouvelles coordonnées du petit cube

*a := x;*

*b := x·y + (1 - x) · (0<sup>z+y+1 mod 2</sup> + y) mod 2;*

*c := x·z + (1 - x) · (0<sup>(z+y) mod 2</sup> + z) mod 2;*

#On détermine les nouvelles orientations des couleurs du petit cube

*d := (x·0 + (1 - x) · (y + z mod 2) · 2 + (1 - x) · 0<sup>y+z mod 2</sup> · 1) mod 3;*

```


$$e := (x \cdot 1 + (1 - x) \cdot (y + z \bmod 2) \cdot 0 + (1 - x) \cdot 0^{y+z \bmod 2} \cdot 2) \bmod 3;$$


$$f := (x \cdot 2 + (1 - x) \cdot (y + z \bmod 2) \cdot 1 + (1 - x) \cdot 0^{y+z \bmod 2} \cdot 0) \bmod 3;$$

#On renvoie les nouvelles données du cube

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix}, \begin{bmatrix} C[d] \\ C[e] \\ C[f] \end{bmatrix};$$

end proc:

```

**\*Fonction qui applique celle qui précède à tous les petits cubes du Rubik's cube et renvoie le nouveau:**

```

RotationBlocL :=proc(Rubix1,$)
local k, Rubix2;
#On change tous les petits cubes du Rubik's cube considéré
for k to 8 do
Rubix2[k] := RotationCubeL(Rubix1[k]);
end do;
Rubix2;
end proc:

```

**\*Fonction qui fait cette fois-ci l'opération dans le sens indirect:**

```

RotationBlocL" :=proc(Rubix1,$)
local k, Rubix2;
#On change tous les petits cubes du Rubik's considéré
for k to 8 do
Rubix2[k]
:= RotationCubeL(RotationCubeL(RotationCubeL(Rubix1[k]))));
end do;
Rubix2;
end proc:

```

**\*Notation:**

Compte tenu de l'utilisation plus que récurrente de ces deux fonctions pour la saisie des mouvements, on les nomme également, jusqu'à la fin du projet, de manière moins explicite mais plus simple par:

```

L := RotationBlocL;
L" := RotationBlocL" ;

```

## B) Selon la face droite

**\*Fonction qui change les données d'un petit cube ayant subi une rotation du bloc R (right) (dans le sens direct):**

```

RotationCubeR := proc(Cube,$)
local x, y, z, C, a, b, c, d, e, f;
#On récupère les coordonnées du petit cube donné
x := Cube[coor][1];
y := Cube[coor][2];
z := Cube[coor][3];
#On récupère l'orientation des couleurs du petit cube donné

```

```

C[0] := Cube[coul][1];
C[1] := Cube[coul][2];
C[2] := Cube[coul][3];
#On détermine les nouvelles coordonnées du petit cube
a := x;
b := (1 - x) · y + x · (0z+y+1 mod 2 + y) mod 2;
c := (1 - x) · z + x · (0(z+y) mod 2 + z) mod 2;
#On détermine les nouvelles orientations des couleurs du petit cube
d := ((1 - x) · 0 + x · (y + z mod 2) · 2 + x · 0y+z mod 2 · 1) mod 3;
e := ((1 - x) · 1 + x · (y + z mod 2) · 0 + x · 0y+z mod 2 · 2) mod 3;
f := ((1 - x) · 2 + x · (y + z mod 2) · 1 + x · 0y+z mod 2 · 0) mod 3;
#On renvoie les nouvelles données du cube
[[ a ] [ C[d] ]
 [ b ] [ C[e] ]
 [ c ] [ C[f] ]];
end proc;

```

**\*Fonction qui applique celle qui précède à tous les petits cubes du Rubik's cube et renvoie le nouveau:**

```

RotationBlocR :=proc(Rubix1,$)
local k, Rubix2;
#On change tous les petits cubes du Rubik's considéré
for k to 8 do
Rubix2[k] := RotationCubeR(Rubix1[k]);
end do;
Rubix2;
end proc;

```

**\*Fonction qui fait cette fois-ci l'opération dans le sens indirect:**

```

RotationBlocR" :=proc(Rubix1,$)
local k, Rubix2;
#On change tous les petits cubes du Rubik's considéré
for k to 8 do
Rubix2[k]
:= RotationCubeR(RotationCubeR(RotationCubeR(Rubix1[k])) );
end do;
Rubix2;
end proc;

```

**\*Notation:**

Compte tenu de l'utilisation plus que récurrente de ces deux fonctions pour la saisie des mouvements, on les nomme également, jusqu'à la fin du projet, de manière moins explicite mais plus simple par:

```

R := RotationBlocR;
R" := RotationBlocR";

```

## C) Selon la face basse

\*Fonction qui change les données d'un petit cube ayant subi une rotation du bloc B (basse) (dans le sens direct):

```

RotationCubeB := proc(Cube,$)
    local x, y, z, C, a, b , c, d, e, f;
    #On récupère les coordonnées du petit cube donné
    x := Cube[coor][1];
    y := Cube[coor][2];
    z := Cube[coor][3];
    #On récupère l'orientation des couleurs du petit cube donné
    C[0] := Cube[coul][1];
    C[1] := Cube[coul][2];
    C[2] := Cube[coul][3];
    #On détermine les nouvelles coordonnées du petit cube
    a := y·x + (1 - y)·(x + 0z) mod 2;
    b := y;
    c := y·z + (1 - y)·(x + z + 0x) mod 2;
    #On détermine les nouvelles orientations des couleurs du petit cube
    d := (y·0 + (1 - y)·z·1 + (1 - y)·(1 - z)·2) mod 3;
    e := (y·1 + (1 - y)·z·2 + (1 - y)·(1 - z)·0) mod 3;
    f := (y·2 + (1 - y)·z·0 + (1 - y)·(1 - z)·1) mod 3;
    #On renvoie les nouvelles données du cube
    [[ a ]], [[ C[d] ]];
    [[ b ]], [[ C[e] ]];
    [[ c ]], [[ C[f] ]]
end proc;

```

\*Fonction qui applique celle qui précède à tous les petits cubes du Rubik's cube et renvoie le nouveau:

```

RotationBlocB :=proc(Rubix1,$)
    local k, Rubix2;
    #On change tous les petits cubes du Rubik's considéré
    for k to 8 do
        Rubix2[k] := RotationCubeB(Rubix1[k]);
    end do;
    Rubix2;
end proc;

```

\*Fonction qui fait cette fois-ci l'opération dans le sens indirect:

```

RotationBlocB" :=proc(Rubix1,$)
    local k, Rubix2;
    #On change tous les petits cubes du Rubik's considéré
    for k to 8 do
        Rubix2[k]
        := RotationCubeB(RotationCubeB(RotationCubeB(Rubix1[k])));
    end do;
    Rubix2;
end proc;

```

\*Notation:

Compte tenu de l'utilisation plus que récurrente de ces deux fonctions pour la saisie des

mouvement, on les nomme également, jusqu'à la fin du projet, de manière moins explicite mais plus simple par:

*B := RotationBlocB;  
B" := RotationBlocB":*

## D) Selon la face haut

\*Fonction qui change les données d'un petit cube ayant subi une rotation du bloc H(haut) (dans le sens direct):

```
RotationCubeH := proc(Cube,$)
    local x, y, z, C, a, b, c, d, e, f;
    #On récupère les coordonnées du petit cube donné
    x := Cube[coor][1];
    y := Cube[coor][2];
    z := Cube[coor][3];
    #On récupère l'orientation des couleurs du petit cube donné
    C[0] := Cube[coul][1];
    C[1] := Cube[coul][2];
    C[2] := Cube[coul][3];
    #On détermine les nouvelles coordonnées du petit cube
    a := (1 - y) · x + y · (x + 0z) mod 2;
    b := y;
    c := (1 - y) · z + y · (x + z + 0x) mod 2;
    #On détermine les nouvelles orientations des couleurs du petit cube
    d := ((1 - y) · 0 + y · z · 2 + y · (1 - z) · 1) mod 3;
    e := ((1 - y) · 1 + y · z · 0 + y · (1 - z) · 2) mod 3;
    f := ((1 - y) · 2 + y · z · 1 + y · (1 - z) · 0) mod 3;
    #On renvoie les nouvelles données du cube
    
$$\begin{bmatrix} a \\ b \\ c \end{bmatrix}, \begin{bmatrix} C[d] \\ C[e] \\ C[f] \end{bmatrix};$$

end proc;
```

\*Fonction qui applique celle qui précède à tous les petits cubes du Rubik's cube et renvoie le nouveau:

```
RotationBlocH := proc(Rubix1,$)
    local k, Rubix2;
    #On change tous les petits cubes du Rubik's considéré
    for k to 8 do
        Rubix2[k]
        := RotationCubeH(RotationCubeH(RotationCubeH(Rubix1[k])));
    end do;
    Rubix2;
end proc;
```

\*Fonction qui fait cette fois-ci l'opération dans le sens indirect:

```
RotationBlocH" := proc(Rubix1,$)
    local k, Rubix2;
```

```

#On change tous les petits cubes du Rubik's considéré
for k to 8 do
    Rubix2[k] := RotationCubeH(Rubix1[k]);
end do;
Rubix2;
end proc:

```

**\*Notation:**

Compte tenu de l'utilisation plus que récurrente de ces deux fonctions pour la saisie des mouvements, on les nomme également, jusqu'à la fin du projet, de manière moins explicite mais plus simple par:

$H := \text{RotationBlocH};$   
 $H'' := \text{RotationBlocH}'';$

## E) Selon la face postérieure

**\*Fonction qui change les données d'un petit cube ayant subi une rotation du bloc P (postérieur) (dans le sens direct):**

```

RotationCubeP := proc(Cube,$)
    local x, y, z, C, a, b , c;
#On récupère les coordonnées du petit cube donné
    x := Cube[coor][1];
    y := Cube[coor][2];
    z := Cube[coor][3];
#On récupère l'orientation des couleurs du petit cube donné
    C[0] := Cube[coul][1];
    C[1] := Cube[coul][2];
    C[2] := Cube[coul][3];
#On détermine les nouvelles coordonnées du petit cube
    a := (x + z mod 2) · x + (1 - (x + z mod 2)) · (x + z + 0x+y+z mod 2
mod 2);
    b := (x + z mod 2) · y + (1 - (x + z mod 2)) · (y + x + 1
    + 0x+y+z mod 2 mod 2);
    c := (x + z mod 2) · z + (1 - (x + z mod 2)) · (x + z + 0x+y+z mod 2
mod 2);

```

#On renvoie les nouvelles données du cube

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix}, \begin{bmatrix} C[0] \\ C[1] \\ C[2] \end{bmatrix};$$

**end proc:**

**\*Fonction qui applique celle qui précède à tous les petits cubes du Rubik's cube et renvoie le nouveau:**

```

RotationBlocP := proc(Rubix1,$)
    local k, Rubix2;

```

#On change tous les petits cubes du Rubik's considéré
**for** k **to** 8 **do**

```

    Rubix2[k]
:= RotationCubeP(RotationCubeP(RotationCubeP(Rubix1[k])));
end do;
Rubix2;
end proc:

```

**\*Fonction qui fait cette fois-ci l'opération dans le sens indirect:**

```

RotationBlocP" :=proc(Rubix1,$)
    local k, Rubix2;
#On change tous les petits cubes du Rubik's considéré
    for k to 8 do
        Rubix2[k] := RotationCubeP(Rubix1[k]);
    end do;
    Rubix2;
end proc:

```

**\*Notation:**

Compte tenu de l'utilisation plus que récurrente de ces deux fonctions pour la saisie des mouvement, on les nomme également, jusqu'à la fin du projet, de manière moins explicite mais plus simple par:

```

P := RotationBlocP ;
P" := RotationBlocP" ;

```

## F) Selon la face avant

**\*Fonction qui change les données d'un petit cube ayant subi une rotation du bloc A (avant) (dans le sens direct):**

```

RotationCubeA := proc(Cube,$)
    local x, y, z, C, a, b , c;
#On récupère les coordonnées du petit cube donné
    x := Cube[coor][1];
    y := Cube[coor][2];
    z := Cube[coor][3];

```

```

#On récupère l'orientation des couleurs du petit cube donné
    C[0] := Cube[coul][1];
    C[1] := Cube[coul][2];
    C[2] := Cube[coul][3];

```

```

#On détermine les nouvelles coordonnées du petit cube
    a := (1 - (x + z mod 2)) · x + (x + z mod 2) · (x + z + 0x+y+z mod 2
mod 2);
    b := (1 - (x + z mod 2)) · y + (x + z mod 2) · (x + y + 0x+y+z mod 2
mod 2);
    c := (1 - (x + z mod 2)) · z + (x + z mod 2) · (x + z + 1 + 0x+y+z mod 2
mod 2);

```

#On renvoie les nouvelles données du cube

```


$$\begin{bmatrix} a \\ b \\ c \end{bmatrix}, \begin{bmatrix} C[0] \\ C[1] \\ C[2] \end{bmatrix};$$

end proc:

```

\*Fonction qui applique celle qui précède à tous les petits cubes du Rubik's cube et renvoie le nouveau:

```

RotationBlocA :=proc(Rubix1,$)
  local k, Rubix2;
  #On change tous les petits cubes du Rubik's considéré
  for k to 8 do
    Rubix2[k] := RotationCubeA(Rubix1[k]);
  end do;
  Rubix2;
end proc:

```

\*Fonction qui fait cette fois-ci l'opération dans le sens indirect:

```

RotationBlocA" :=proc(Rubix1,$)
  local k, Rubix2;
  #On change tous les petits cubes du Rubik's considéré
  for k to 8 do
    Rubix2[k]
    := RotationCubeA(RotationCubeA(RotationCubeA(Rubix1[k])));
  end do;
  Rubix2;
end proc:

```

Notation:

Compte tenu de l'utilisation plus que récurrente de ces deux fonctions pour la saisie des mouvements, on les nomme également, jusqu'à la fin du projet, de manière moins explicite mais plus simple par:

```

A := RotationBlocA;
A" := RotationBlocA";

```

## III) Les outils

Principalement pour la représentation du Rubik's cube.

### A) Le patron

Pour dessiner le patron, on dessine les petits cubes un par un sur un même plot.

\*Fonction qui dessine un petit carré:

```

DessinePetitCube :=proc(Cube,$)
  local S, x, y, t, C;
  #On récupère les coordonnées et les couleurs à adopter sur le graphique.
  t := 0^1 - Cube[coor][3].(1 + 2·Cube[coor][2]) + 0^Cube[coor][3].(2 + 2
  ·Cube[coor][2]);

```

```

 $x := 4 \cdot (0^2 - \text{Cube}[coor][1] - \text{Cube}[coor][3] - 0^{\text{Cube}[coor][1] + \text{Cube}[coor][3]});$ 
 $y := 0;$ 
 $C := \text{Cube}[coul];$ 
#On dessine les 3 petits carrés correspondant au petit cube donné.
if  $t \leq 2$  then
   $S := \text{polygonplot}([ [x, y], [x + (-1)^t, y], [x + (-1)^t, y - 1], [x, y - 1], [x, y] ], \text{color} = C[2]);$ 
   $S := S, \text{polygonplot}([ [x - 0^{t-1}, y - 0^{2-t}], [x - 0^{t-1} + (-1)^t, y - 0^{2-t}], [x - 0^{t-1} + (-1)^t, y - 0^{2-t} - 1], [x - 0^{t-1}, y - 0^{2-t} - 1], [x - 0^{t-1}, y - 0^{2-t}], \text{color} = C[3]);$ 
   $S := S, \text{polygonplot}([ [x + 0^{2-t}, y - 0^{t-1}], [x + 0^{2-t} + (-1)^t, y - 0^{t-1}], [x + 0^{2-t} + (-1)^t, y - 0^{t-1} - 1], [x + 0^{2-t}, y - 0^{t-1} - 1], [x + 0^{2-t}, y - 0^{t-1}], \text{color} = C[1]);$ 
else
   $S := \text{polygonplot}([ [x, y], [x + (-1)^t, y], [x + (-1)^t, y + 1], [x, y + 1], [x, y] ], \text{color} = C[2]);$ 
   $S := S, \text{polygonplot}([ [x + 0^{4-t}, y + 0^{t-3}], [x + 0^{4-t} + (-1)^t, y + 0^{t-3}], [x + 0^{4-t} + (-1)^t, y + 0^{t-3} + 1], [x + 0^{4-t}, y + 0^{t-3} + 1], [x + 0^{4-t}, y + 0^{t-3}], \text{color} = C[3]);$ 
   $S := S, \text{polygonplot}([ [x - 0^{t-3}, y + 0^{4-t}], [x - 0^{t-3} + (-1)^t, y + 0^{4-t}], [x - 0^{t-3} + (-1)^t, y + 0^{4-t} + 1], [x - 0^{t-3}, y + 0^{4-t} + 1], [x - 0^{t-3}, y + 0^{4-t}], \text{color} = C[1]);$ 
end if;
   $S;$ 
end proc:

```

#### \*Fonction qui trace le patron d'un Rubix:

*DessinePatron :=proc(Rubix)*

#On dessine en même temps les 8 petits cubes.

```

display(DessinePetitCube(Rubix[1]), DessinePetitCube(Rubix[2]),
        DessinePetitCube(Rubix[3]), DessinePetitCube(Rubix[4]),
        DessinePetitCube(Rubix[5]), DessinePetitCube(Rubix[6]),
        DessinePetitCube(Rubix[7]), DessinePetitCube(Rubix[8]), view = [-4 .. 4, -4 .. 4],
        title = "Patron du cube", axes = none);
end proc:

```

## B)Le volume

Ceci est la deuxième version de la représentation 3D (i.e. avec les mouvements). Son principe: afficher les 8 petits cubes du Rubik's séparément en indiquant pour chacun, les coordonnées de leurs 8 sommets, afin de pouvoir les faire pivoter individuellement pour afficher le mouvement.

#### \*Déclarations des coordonées références (en vue des les faire pivoter):

Il s'agit des coordonnées par défaut d'un Rubik's (sans mouvement). Ce sont des références et ne sont donc pas modifiées.

*k := 0 :*

# k compte le nombre totale de coordonnées.

```

for  $y1$  from -1 to 1 do
  for  $z1$  from -1 to 1 do
    for  $x1$  from -1 to 1 do
       $k := k + 1; P1[k] := [x1, y1, z1];$ 
    end do; end do; end do;

```

### \*Fonction effectuant une rotation d'une face et d'un angle donné:

On lui envoie des coordonnées (M1) ainsi que la face à laquelle il appartient et l'angle et il renvoie les nouvelles coordonnées correspondantes à la nouvelle position après la rotation.

```
Rotation :=proc(θ,face,M1,$)
    local k, M2;
    M2 := M1;
    if face = L or face = R then k := 1
    elif face = A or face = P then k := 2
    elif face = B or face = H then k := 3 end if;
    M2[k mod 3 + 1] := M1[k mod 3 + 1]·cos(θ) - sin(θ)·M1[k + 1 mod 3 + 1];
    M2[k + 1 mod 3 + 1] := M1[k mod 3 + 1]·sin(θ) + cos(θ)·M1[k + 1 mod 3
    + 1];
    M2;
end proc;
```

### \*Fonction qui renvoie les plot d'un petit cube donné:

Vu qu'il s'agit de la version II, il faut rentrer, en plus des couleurs, les coordonnées des 8 sommets.

```
DessineCube :=proc(A, B, C, D, E, F, G, H, c1, c2, c3, c4, c5, c6,$)
    local S;
    S := polygonplot3d([A, B, D, C], color = c1);
    S := S, polygonplot3d([B, F, H, D], color = c2);
    S := S, polygonplot3d([E, F, H, G], color = c3);
    S := S, polygonplot3d([A, E, G, C], color = c4);
    S := S, polygonplot3d([A, B, F, E], color = c5);
    S := S, polygonplot3d([C, D, H, G], color = c6);
end proc;
```

### \*Fonction qui dessine un Rubik's cube donné:

Il faut lui donner un triplet [Rubik's cube, Angle de rotation, Face à pivoter]

```
DessineVolume :=proc(V,$)
```

```
    local i, k, S, PT, Rubix, θ, face;
```

```
#On récupère les trois données
```

```
Rubix := V[1]; θ := V[2]; face := V[3];
```

```
#A l'aide des coordonnées de références, on définit les coordonnées de départ de chaque
petit cube`:
```

```
PT[1] := [P1[1], P1[2], P1[4], P1[5], P1[10], P1[11], P1[13],
P1[14]];
PT[2] := [P1[2], P1[3], P1[5], P1[6], P1[11], P1[12], P1[14],
P1[15]];
PT[3] := [P1[4], P1[5], P1[7], P1[8], P1[13], P1[14], P1[16],
P1[17]];
PT[4] := [P1[5], P1[6], P1[8], P1[9], P1[14], P1[15], P1[17],
P1[18]];
PT[5] := [P1[10], P1[11], P1[13], P1[14], P1[19], P1[20], P1[22],
P1[23]];
PT[6] := [P1[11], P1[12], P1[14], P1[15], P1[20], P1[21], P1[23],
P1[24]];
PT[7] := [P1[13], P1[14], P1[16], P1[17], P1[22], P1[23], P1[25],
P1[26]];
```

```

PT[8] := [P1[14], P1[15], P1[17], P1[18], P1[23], P1[24], P1[26],
P1[27]]:
#En fonction de la face qui tourne, on désigne les petits cubes à faire bouger.
    if face = A then S := [1, 2, 3, 4]
    elif face = P then S := [5, 6, 7, 8]
    elif face = B then S := [1, 2, 5, 6]
    elif face = H then S := [3, 4, 7, 8]
    elif face = R then S := [2, 4, 6, 8]
    elif face = L then S := [1, 3, 5, 7] end if;
#On fait bouger les petits cubes concernés.
    for k in S do
        for i to 8 do
            PT[k][i] := Rotation(θ, face, PT[k][i]);
        end do;
    end do;
#On affiche le résultat.
display( DessineCube( (PT[1])[], Rubix[1][coul][2], black, black, Rubix[1][coul][3],
Rubix[1][coul][1], black),
DessineCube( (PT[2])[], Rubix[2][coul][2], Rubix[2][coul][1], black, black,
Rubix[2][coul][3], black),
DessineCube( (PT[3])[], Rubix[3][coul][2], black, black, Rubix[3][coul][1],
black, Rubix[3][coul][3]),
DessineCube( (PT[4])[], Rubix[4][coul][2], Rubix[4][coul][3], black, black,
black, Rubix[4][coul][1]),
DessineCube( (PT[5])[], black, black, Rubix[5][coul][2], Rubix[5][coul][1],
Rubix[5][coul][3], black),
DessineCube( (PT[6])[], black, Rubix[6][coul][3], Rubix[6][coul][2], black,
Rubix[6][coul][1], black),
DessineCube( (PT[7])[], black, black, Rubix[7][coul][2], Rubix[7][coul][3],
black, Rubix[7][coul][1]),
DessineCube( (PT[8])[], black, Rubix[8][coul][1], Rubix[8][coul][2], black,
black, Rubix[8][coul][3]),
projection = 0.6, orientation = [-52, 60], title = "Représentation 3D du cube", view = [-1.5
..1.5, -1.5 ..1.5, -1.5 ..1.5]);
end proc;

```

## C) La création d'un Rubik's cube

\*Fonction qui crée un Rubik's cube à partir de ses couleurs données dans l'ordre de référence:

Il faut rentrer les 24 couleurs du Rubik's cube dans l'ordre déterminé.

```
CreerRubix :=proc(c1, c2, c3, c4, c5, c6, c7, c8, c9, c10, c11, c12, c13, c14, c15, c16, c17,
c18, c19, c20, c21, c22, c23, c24,$)
```

```


$$\left[ \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} C[c19] \\ C[c1] \\ C[c14] \end{bmatrix} \right], \left[ \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} C[c5] \\ C[c2] \\ C[c20] \end{bmatrix} \right], \left[ \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}, \begin{bmatrix} C[c16] \\ C[c3] \\ C[c21] \end{bmatrix} \right], \left[ \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} C[c22] \\ C[c4] \\ C[c7] \end{bmatrix} \right],$$


$$\left[ \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} C[c13] \\ C[c10] \\ C[c17] \end{bmatrix} \right], \left[ \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} C[c18] \\ C[c9] \\ C[c6] \end{bmatrix} \right], \left[ \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} C[c23] \\ C[c12] \\ C[c15] \end{bmatrix} \right], \left[ \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, \begin{bmatrix} C[c8] \\ C[c11] \\ C[c24] \end{bmatrix} \right]$$

end proc:

```

#### \*Fonction qui crée aléatoirement un Rubik's cube:

Il s'agit seulement de partir du Rubik's fini et de lui faire subir des mouvements aléatoires pour le mélanger.

```
CreerRubixAleatoire :=proc( {n :: integer := 30})
```

# n désigne le nombre de coups que l'on exécute.

```
local M, Rubix, k;
```

```
Rubix := RUBIX.FINI;
```

#On initialise les différents mouvements possibles.

```
M[1] := A;
```

```
M[2] := A";
```

```
M[3] := P;
```

```
M[4] := P";
```

```
M[5] := B;
```

```
M[6] := B";
```

```
M[7] := H;
```

```
M[8] := H";
```

```
M[9] := B;
```

```
M[10] := B";
```

```
M[11] := R;
```

```
M[12] := R";
```

```
M[13] := L;
```

```
M[14] := L";
```

#On effectue n fois des opérations élémentaires sur le Rubix de référence

```
for k to n do
```

```
Rubix := Rafraichir(BougerRubix(Rubix, M[rand(1..14)( )], patron =false, cube=true, volume=false));
```

```
end do;
```

```
Rubix;
```

```
end proc:
```

## E)Déplacement du Rubik's Cube

#### \*Fonction qui exécute une série de mouvements sur un Rubik's cube:

Il s'agit toujours de la version II, forcément plus lente que la I, mais bien plus esthétique et claire.

```
BougerRubix :=proc( { volume :: boolean := true, cube :: boolean := false, image :: integer := 3, patron :: boolean := false})
```

# volume indique si l'on doit afficher le volume

# cube indique si l'on renvoie le Rubik's cube après avoir subi les mouvements

```

# image indique le nombre d'images à afficher pour 1 mouvement
# patron indique si l'on doit afficher le patron
local k, Rubix, seq, seq2, i, j, p, n, ε, k2, x;
#On initialise les variables
    #Le premier argument est le Rubik's cube considéré
    Rubix := op(1, [args]);
    #Indice pour stocker les différents états du Rubik's cube
    i := 1; x := 1;
    #Indice pour compter le nombre d'options entrées
    p := 0;
    seq[0] := [Rafraichir(Rubix), 0, 0];
    seq2[0] := Rafraichir(Rubix);
#On effectue toutes les opérations demandées
    for k in [op(2 ..nops([args]), [args])] do
        #On ne fait rien si c'est une option
        if type(k, boolean) or type(k, posint) then p := p + 1 else
            # On prend le sens de la rotation en fonction de la face considérée.
            if k=A" or k=P or k=B" or k=H or k=R or k=L"
                then ε := 1; else ε := -1; end if,
                # S'il s'agit d'un rotation inverse, on prend la rotation normale et on inverse le sens.
                if k=A" then k2 := A
                elif k=B" then k2 := B
                elif k=H" then k2 := H
                elif k=P" then k2 := P
                elif k=R" then k2 := R
                elif k=L" then k2 := L
                else k2 := k; end if;
                # On enregistre les `image` images du mouvement.
                for j from 1 to image do
                    seq[i] := [Rafraichir(Rubix), ε·j·Pi/2, k2];
                    i := i + 1;
                end do;
                # Pour le patron
                seq2[x] := Rafraichir(Rubix);
                x := x + 1;
            end if;
        end if;
    end do;
    # On rappelle que les mouvements (A,A",P,P",..) sont des procédures d'argument un
    # Rubik's cube.
    Rubix := k(Rubix);
    end if;
    end do;
    seq2[x] := Rafraichir(Rubix);
#On renvoie ce que l'on a demandé
    if patron then print/animate(DessinePatron, [seq2[n]], n = [ '$'(0 ..x)],
    paraminfo=false )) end if;
        if volume then print/animate(DessineVolume, [seq[n]], n = [ '$'(0 ..i - 1)],
        paraminfo=false )) end if;
            if cube then Rubix; end if;
        end proc;

```

## IV) Première méthode de résolution

### A) Les mouvements clefs

#### \*Mouvement de base:

Qui échanger la position du bloc 1 et 2 et l'orientation des blocs 1, 2 et 3  
*MouvementDeBase := L", H", L", H", R, H, L, H, L, B, B* : On l'abrége:  
*mb := MouvementDeBase* :

#### \*Mouvement d'échange:

Mouvement qui échange le petit cube 1 et 2 sans en changer l'orientation  
*MouvementDEchange := mb, mb, mb* : On l'abrége:  
*me := MouvementDEchange* :

#### \*Mouvement d'orientation:

Mouvement qui change l'orientation du petit cube 1 et 2.  
*MouvementDOrientation := B, mb, mb, B, mb, B", mb, mb, B, mb, B, B* :  
On l'abrége:  
*mo := MouvementDOrientation* :

### B) Echange de 2 petits cubes

#### \*Mouvement pour échanger un seul petit cube avec le n°1:

```
MouvementPourEchanger1 := proc(a,$)
    local coord, s;
    coord := NumeroVersCoordonnees(a);
#On traite chaque cas différent
    if coord = [1, 0, 1] then s := R", me, R;
    elif coord = [0, 1, 0] then s := P", B", me, B, P;
    elif coord = [0, 1, 1] then s := A, me, A";
    elif coord = [1, 1, 0] then s := R, me, R";
    elif coord = [1, 1, 1] then s := R, R, me, R, R;
    elif coord = [0, 0, 0] then s := B", me, B;
    elif coord = [1, 0, 0] then s := me;
    elif coord = [0, 0, 1] then s := NULL : end if;
#On renvoie la liste des mouvement à faire
    s;
end proc;
```

#### \*Mouvement pour échanger n'importe quel cube entre eux:

```
MouvementPourEchanger := proc(a, b,$)
    MouvementPourEchanger1(a), MouvementPourEchanger1(b),
    MouvementPourEchanger1(a);
end proc;
```

On l'abrége:

*mpe := MouvementPourEchanger* :

## C)Changement de l'orientation de 2 petits cubes

### \*Mouvement pour changer l'orientation d'un petit cube et du n°1:

```
MouvementPourOrienter1 :=proc(a,$)
    local coord, s;
    coord := NumeroVersCoordonnees(a);
#On traite chaque cas différent
    if coord = [1, 0, 1] then s := R", mo, R;
    elif coord = [0, 1, 0] then s := P", B", mo, B, P;
    elif coord = [0, 1, 1] then s := A, mo, A";
    elif coord = [1, 1, 0] then s := R, mo, R";
    elif coord = [1, 1, 1] then s := R, R, mo, R, R;
    elif coord = [0, 0, 0] then s := B", mo, B;
    elif coord = [1, 0, 0] then s := mo;
    elif coord = [0, 0, 1] then s := NULL : end if;
#On renvoie la liste des mouvement à faire
    s
end proc;
```

### \*Mouvement pour échanger n'importe quel cube entre eux:

```
MouvementPourOrienter :=proc(a, b,$)
    MouvementPourOrienter1(a), MouvementPourOrienter1(b);
end proc;
```

On l'abrège:

*mpo := MouvementPourOrienter :*

## D)Résolution

### \*Fonction qui place les petits cubes au bon endroit selon leur couleur:

```
ResoudreCoordonnees :=proc(Rubix1,$)
    local Rubix2, s, k, temp, a, b;
    s := NULL;
    Rubix2 := Rubix1;
#On règle tout ça petit cube par petit cube
    for k to 8 do
        #On vérifie s'il y a raison de bouger
        a := CouleursVersNumero(      Rubix2[k][coul][1],
        Rubix2[k][coul][2], Rubix2[k][coul][3]);
        b := CoordonneesVersNumero(Rubix2[k][coor][1],
        Rubix2[k][coor][2], Rubix2[k][coor][3]);
        if a ≠ b then temp := mpe(a, b);
        s := s, temp;
        Rubix2 := BougerRubix(Rubix2, temp, patron = false,
        volume = false, cube = true);
        end if; end do;
        s;
end proc;
```

### \*Fonction qui oriente les petits cubes correctement:

```
ResoudreOrientations :=proc(Rubix1,$)
    local Rubix2, s, temp, i, j, k, a, b, c;
    s := NULL;
```

```

    Rubix2 := Rafraichir(Rubix1);
#On règle tout ça petit cube par petit cube
    for i from 1 to 8 do
        a := EstCycle(Rafraichir(Rubix2)[i][coul]);
    for j from i + 1 to 8 do
        b := EstCycle(Rafraichir(Rubix2)[j][coul]);
    if b ≠ 0 then
        if a = 1 then temp := mpo(i, j); s := s, temp; i := 0; j := 9;
            Rubix2 := Rafraichir(BougerRubix(Rubix2, temp,
            patron = false, volume = false, cube = true));
        end if;
        if a = 2 then temp := mpo(j, i); s := s, temp; i := 0; j := 9;
            Rubix2 := Rafraichir(BougerRubix(Rubix2, temp,
            patron = false, volume = false, cube = true));
        end if;
    end if; end do; end do;
    s;
end proc;

```

#### \*Fonction tant attendu de résolution du rubick's cube:

```

ResoudreI" :=proc(Rubix1,$)
    local Rubix2, s, temp, k;
    Rubix2 := Rubix1;
#On vérifie que le cube est bien résoluble
    if not (Axiome1(Rubix2) and Axiome2et3(Rubix2)) then
        print("Cube non résoluble") else
#On résoud les coordonnées
        s := ResoudreCoordonnees(Rubix2);
        Rubix2 := Rafraichir(BougerRubix(Rubix2, s, patron = false, volume
        = false, cube = true));
#On résoud les orientations
        s := s, ResoudreOrientations(Rubix2);
    end if; end proc;

```

#### \*Fonction finale de résolution d'un rubix crée:

```

ResoudreI :=proc(Rubix, {nbimage :: posint := 1, vol :: boolean := true, pat :: boolean
:= false, commandes :: boolean := false, nbcoups :: boolean := true},$)
    local Mouvement:
    Mouvement := ResoudreI"(Rubix);
    if nbcoups then printf("Cube résolu en %g coups", nops([Mouvement])); end if;
    BougerRubix(Rubix, Mouvement, image = nbimage, volume = vol, patron = pat);
    if commandes then Mouvement; end if;
end proc;

```

## E) Tentative (vaine) de simplification

Vu le nombre d'opération nécessaire pour résoudre le rubick's cube avec la méthode précédent, on tente de faire un programme qui les simplifirait.

#### \*Fonction de teste de l'égalité de deux rubick's cube:

```

EstEgale :=proc(Rub1, Rub2)
    local Rubix1, Rubix2, Bool, k, c;

```

```

Bool := true;
Rubix1 := Rafraichir(Rub1);
Rubix2 := Rafraichir(Rub2);

for k to 8 do
  for c to 3 do
    Bool := Bool and evalb(Rubix1[k][coul][c] = Rubix2[k][coul][c])
  end do; end do;
  Bool;
end proc:

```

#### \*Fonction de simplification d'une suite de mouvement:

```

Simplifier1 :=proc( )
  local mouv, k, l, i; i := 0; k := 1;
  mouv := [args];
  l := nops(mouv);
  for k while k < l - 2 do
    if EstEgale(Rafraichir(BougerRubix(rubix, mouv[k], mouv[k + 1], patron
=false, volume=false, cube=true)), rubix) then i := i + 1;

    mouv := [op(1..k - 1, mouv), op(k + 2..l, mouv)]; end if;
    l := nops(mouv);
  end do; k := 1;
  for k while k < l - 4 do
    if EstEgale(Rafraichir(BougerRubix(rubix, mouv[k], mouv[k + 1], mouv[k
+ 2], mouv[k + 3], patron=false, volume=false, cube=true)), rubix) then i := i + 1;

    mouv := [op(1..k - 1, mouv), op(k + 4..l, mouv)]; end if;
    l := nops(mouv);
  end do;
  op(1..l, mouv), nops(mouv);
end proc:

```

## V) Deuxième méthode de résolution

Cette méthode se base sur un tri progressif des différents cas rencontrés dans la résolution du Rubik's cube (selon le guide officiel).

### A) Phase 1

Elle consiste à faire la première face du cube

#### \*Fonction qui donne la position d'un petit cube dans un Rubik's cube donné à partir de ses couleurs:

Contrairement à celle qui est dans les outils, celle-ci recherche le triplet de couleurs, non plus dans le Rubik's cube fini mais dans n'importe lequel.

```

OuEstPetitCube :=proc(Rubix, c1, c2, c3,$)
  local k;
  for k to 8 do
    if {Rubix[k][coul][1], Rubix[k][coul][2], Rubix[k][coul][3]}

```

```
= {c1, c2, c3} then return k; end if;
    end do;
end proc;
```

## ***α) Phase 1-a***

Elle consiste à placer côté à côté les deux premiers carrés

### **\*Fonction qui place le 1er cube voulu à l'endroit voulu:**

Techniquement, elle renvoie les mouvements pour placer un petit cube en 4ème position.

```
ChangerDeCoin4 :=proc(a,$)
```

```
    if a = 1 then A, P", A, P";
    elif a = 2 then A, P";
    elif a = 3 then A", P;
    elif a = 4 then NULL;
    elif a = 5 then B", H, A, P", A, P";
    elif a = 6 then B, H", A, P";
    elif a = 7 then H, B", H, B";
    elif a = 8 then H", B end if;
end proc;
```

### **\*Fonction qui oriente le cube précédent:**

Il s'agit là d'une étape de résolution vis à vis de la méthode officielle, ce n'est pas un outil utilisable pour autre chose.

```
ChangerDOrientation4 :=proc(Rubix,$)
```

```
    local n;
    n := EstCycleII(Rubix[4][coul], 4);
    if n = 0 then H, B", R, L
    elif n = 1 then H", B, A", P end if;
end proc;
```

### **\*Fonction qui exécute la phase 1-a:**

Cette fonction renvoie les mouvements à faire pour résoudre l'étape 1 phase a du cube.

```
Etape1PhaseA :=proc(Rubix,$)
```

```
    local Rubix2, a, n, S, mouv;
    S := NULL;
```

# On commence d'abord par placer le cube dans la position voulue

```
Rubix2 := Rafraichir(Rubix);
```

```
mouv := ChangerDeCoin4(OuEstPetitCube(Rubix2, C[o], C[w],
C[b]));
```

```
Rubix2 := Rafraichir(BougerRubix(Rubix2, mouv, cube = true, patron
=false, volume = false));
```

```
mouv := mouv, ChangerDOrientation4(Rubix2);
```

```
Rubix2 := Rafraichir(BougerRubix(Rubix2,
```

```
ChangerDOrientation4(Rubix2), cube = true, patron = false, volume = false));
```

# On place ensuite le deuxième petit cube en fonction des différents cas.

```
a := OuEstPetitCube(Rubix2, C[w], C[g], C[o]);
```

```
if a = 1 then S := B";
```

```
elif a = 3 then S := L", B";
```

```
elif a = 5 then S := B;
```

```
elif a = 7 then S := P; end if;
```

```
Rubix2 := Rafraichir(BougerRubix(Rubix2, S, cube = true, patron
=false, volume = false));
```

```
a := OuEstPetitCube(Rubix2, C[w], C[g], C[o]);
```

```
n := EstCycleII(Rubix2[a][coul], a); S := mouv, S;
```

```

if  $a = 2$  and  $n = 2$  then  $S := S, B, L'', B, B, P, B'', P'';$ 
elif  $a = 2$  and  $n = 0$  then  $S := S, B'', P'';$ 
elif  $a = 2$  and  $n = 1$  then  $S := S, B, B, R'', B, R;$ 
elif  $a = 8$  and  $n = 0$  then  $S := S, P, P, R'', B, R;$ 
elif  $a = 8$  and  $n = 2$  then  $S := S, P'', L, P'', P'';$ 
elif  $a = 6$  and  $n = 0$  then  $S := S, P'';$ 
elif  $a = 6$  and  $n = 1$  then  $S := S, P, P, L, P, P;$ 
elif  $a = 6$  and  $n = 2$  then  $S := S, P, R'', B, R;$  end if;  $S;$ 
end proc:

```

## **B) Phase 1-b**

Elle consiste à placer le troisième petit cube à coté des deux premiers petits cubes

### **Fonction qui exécute la phase 1-b:**

Même principe que précédemment.

```

Etape1PhaseB :=proc(Rubix,$)
    local Rubix2, a, n, S;
    S := NULL;
    Rubix2 := Rafraichir(BougerRubix(Rubix, B'', H, cube = true, patron
    =false, volume =false));
    a := OuEstPetitCube(Rubix2, C[r], C[g], C[w]);
    n := EstCycleII(Rubix2[a][coul], a);
# Si le cas n'est pas couvert, on se place dans l'un qui l'est.
    if  $a = 2$  and  $n = 1$  then  $S := B;$ 
    elif  $a = 2$  and  $n = 0$  then  $S := A;$ 
    elif  $a = 2$  and  $n = 0$  then  $S := A;$ 
    elif  $a = 3$  and  $n = 2$  then  $S := A;$ 
    elif  $a = 4$  and  $n = 2$  then  $S := A, A;$ 
    elif  $a = 5$  then  $S := B'';$ 
    elif  $a = 6$  then  $S := B, B;$  end if;
    Rubix2 := Rafraichir(BougerRubix(Rubix2, S, cube = true, patron
    =false, volume =false));
    a := OuEstPetitCube(Rubix2, C[r], C[g], C[w]);
    n := EstCycleII(Rubix2[a][coul], a); S := B'', H, S;
# On traite les différents cas possibles.
    if  $a = 1$  and  $n = 2$  then  $S := S, A'';$ 
    elif  $a = 1$  and  $n = 0$  then  $S := S, B'', L'', B, L;$ 
    elif  $a = 1$  and  $n = 1$  then  $S := S, B'', A, A;$ 
    elif  $a = 2$  and  $n = 2$  then  $S := S, A, A;$ 
    elif  $a = 3$  and  $n = 1$  then  $S := S, A, B'', A, A;$ 
    elif  $a = 3$  and  $n = 0$  then  $S := S, A, B'', L'', B, L;$ 
    elif  $a = 4$  and  $n = 0$  then  $S := S, A, A, B'', L'', B, L;$ 
    elif  $a = 4$  and  $n = 1$  then  $S := S, P, P, H'', P, P;$  end if;  $S;$ 
end proc:

```

## **C) Phase 1-c**

Elle consiste à placer le troisième carré à coté des deux premiers carrés

### **\*Fonction qui exécute la phase 1-c:**

Même principe que précédemment.

```

Etape1PhaseC :=proc(Rubix,$)
    local Rubix2, a, n, S;
    S := NULL;

```

```

    Rubix2 := Rafraichir(Rubix);
    a := OuEstPetitCube(Rubix2, C[w], C[b], C[r]);
    n := EstCycleII(Rubix2[a][coul], a);
# Si le cas n'est pas couvert, on se place dans l'un qui l'est.
    if a = 1 and n = 0 then S := B'', L, A, L'';
    elif a = 2 and n = 0 then S := L, A, L'';
    elif a = 5 then S := B, B, L, A, L'';
    elif a = 6 then S := B, L, A, L''; end if;
    Rubix2 := Rafraichir(BougerRubix(Rubix2, S, cube = true, patron
=false, volume =false));
    a := OuEstPetitCube(Rubix2, C[w], C[b], C[r]);
    n := EstCycleII(Rubix2[a][coul], a);
# On traite les différents cas possibles.
    if a = 1 and n = 2 then S := S, R, B'', R'', A'', B'', A, B, B, R, B'', R'';
    elif a = 1 and n = 1 then S := S, B, B, A'', B, A;
    elif a = 2 and n = 2 then S := S, B'', A'', B, A;
    elif a = 2 and n = 1 then S := S, B, R, B'', R'';
    elif a = 4 and n = 1 then S := S, A'', B'', A, B, B, R, B'', R'';
    elif a = 4 and n = 0 then S := S, R, B, R'', B, B, A'', B, A; end if; S;
end proc;

```

#### \*Fonction qui exécute la phase 1:

Elle cumule simplement les 3 morceaux précédents.

```

Etape1 :=proc(Rubix,$)
local Rubix2, S1, S2, S3;
S1 := Etape1PhaseA(Rubix) ;
Rubix2 := Rafraichir(BougerRubix(Rubix, S1, cube = true, patron =false, volume
=false));
S2 := Etape1PhaseB(Rubix2) ;
Rubix2 := Rafraichir(BougerRubix(Rubix2, S2, cube = true, patron =false, volume
=false));
S3 := Etape1PhaseC(Rubix2) ;
S1, S2, S3;
end proc;

```

## C) Phase 2

Elle consiste à bien placer les autres petits cubes. Toutes les fonctions suivantes sont des tests et outils au service de cette étape, ils ne servent donc à rien d'autre et n'ont individuellement aucune utilité, ils doivent plus être vus comme des procédures internes à étape2.

#### \*Fonction annexe de test d'orientation:

```

Etape2TestOrientation :=proc(Rubix,$)
local k, S;
S := NULL;
for k in [3, 4, 7, 8] do
if EstCycleII(Rubix[k][coul]) = 0 then S := S, k; end if;
end do; S;
end proc;

```

#### \*Fonction annexe de test de position:

```

Etape2TestPlace :=proc(Rubix,$)
local k, S;

```

```

 $S := \text{NULL};$ 
 $\text{for } k \text{ in } [1, 2, 5, 6, 3, 4, 7, 8] \text{ do}$ 
 $\quad \text{if } \text{CouleursVersNumero}(\text{Rubix}[k][\text{coul}][1], \text{Rubix}[k][\text{coul}][2],$ 
 $\quad \quad \text{Rubix}[k][\text{coul}][3]) = k \text{ then } S := S, k; \text{end if};$ 
 $\quad \text{end do; } S;$ 
 $\text{end proc:}$ 

```

**\*Fonction annexe de test de position 2:**

```

Etape2TestPos :=proc(Rubix,$)
    local k, S;
     $S := \text{NULL};$ 
    for k in [3, 4, 8, 7] do
         $S := S, \text{CouleursVersNumero}(\text{Rubix}[k][\text{coul}][1], \text{Rubix}[k][\text{coul}][2],$ 
         $\text{Rubix}[k][\text{coul}][3]);$ 
    end do; [S];
end proc:

```

**\*Fonction annexe de test de position 3:**

```

Etape2TestPosPlace :=proc(v,$)
    local a, b, c, d;
     $a := v[1];$ 
     $b := v[2];$ 
     $c := v[3];$ 
     $d := v[4];$ 
    if a = 3 then [a, b, c, d], [ ]
    elif b = 3 then [b, c, d, a], [H'', B]
    elif c = 3 then [c, d, a, b], [H, B'', H, B'']
    elif d = 3 then [d, a, b, c], [H, B''] end if;
end proc:

```

**\*Définition des mouvements clés:**

L'étape 2 se base sur les quelques mouvements que voici.

*Etape2Mouvement1 := L'', H, H, L, R, H'', L'', H, R'', H, H, L, H;*

*Etape2Mouvement2 := A'', H'', A, H, P, H'', A'', H, A, P'';*

*Etape2Mouvement3 := NULL :*

*Etape2Mouvement5 := P'', H, A, H'', P, H, A'', H, H :*

**\*Bijections utilisées dans l'étape 2:**

Fonctions obscures pourtant indispensables pour l'étape 2...

*Bij1 := x → x - 4 + 2 · 0<sup>6-x</sup> mod 5 :*

*Bij2 := x → x - 1 + 2 · 0<sup>8-x</sup> mod 5 :*

**\*Alignement après l'étape 2:**

Ceci est plus une fonction préparatoire à l'étape 3.

```

Etape2Aligne :=proc(Rubix, $)
    local a;
     $a := [\text{Bij1}(\text{OuEstPetitCube}(\text{Rubix}, \text{NumeroVersCouleurs}(1))),$ 
     $\text{Bij2}(\text{OuEstPetitCube}(\text{Rubix}, \text{NumeroVersCouleurs}(3)))];$ 
     $a := a[1] - a[2];$ 
    if a = 0 then NULL;
    elif a = 1 or a = -3 then H;
    elif a = -1 or a = 3 then H';
    elif a = 2 or a = -2 then H, H; end if;
end proc:

```

### \*Fonction qui fait l'étape 2:

La fonction qui nous intéresse:

```
Etape2 :=proc(Rubix,$)
    local Rubix2, mouv, ETPPETP, Pos;
    mouv[1] := A, P", A, P";
#Les tests sont un peu lourds.
    Rubix2 := Rafraichir(BougerRubix(Rubix, mouv[1], patron = false, volume = false,
    cube = true));
    ETPPETP := [Etape2TestPosPlace(Etape2TestPos(Rubix2))];
    mouv[2] := (ETPPETP[2])[];
    Pos := ETPPETP[1];
#Mais après c'est comme d'habitude, tri des différents cas.
    if Pos = [3, 7, 8, 4] then mouv[3] := Etape2Mouvement1;
    elif Pos = [3, 7, 4, 8] then mouv[3] := Etape2Mouvement2, H, Etape2Mouvement5;
    elif Pos = [3, 4, 8, 7] then mouv[3] := Etape2Mouvement3;
    elif Pos = [3, 8, 7, 4] then mouv[3] := H, Etape2Mouvement5;
    elif Pos = [3, 4, 7, 8] then mouv[3] := H, H, Etape2Mouvement5;
    elif Pos = [3, 8, 4, 7] then mouv[3] := H", Etape2Mouvement5; end if;
    Rubix2 := Rafraichir(BougerRubix(Rubix2, mouv[2], mouv[3], patron = false, volume
    = false, cube = true));
    mouv[1], mouv[2], mouv[3], Etape2Aligne(Rubix2);
end proc;
```

## C)Phase 3

### \*Mouvement clef de l'étape 3:

L'étape 3 ne se base que sur un seul mouvement à bien utiliser.

```
Etape3Mouvement := R, B", R", A", B", A, H", A", B, A, R, B, R", H;
```

### \*Décalage préliminaire pour calculer le cycle:

Cette fonction permet de se placer dans une position où la valeur du cycle est unique ! Extraordinaire. Bien sur, cela ne fonctionne que dans ce cas très précis et cette fonction et la suivante ne fonctionnent pas ailleurs.

```
Etape3ChangementDeRef :=proc(Rubix,$)
    local a;
    a := OuEstPetitCube(Rafraichir(Rubix), white, blue, red);
    if a = 1 then NULL;
    elif a = 2 then B, H";
    elif a = 5 then B", H;
    elif a = 6 then B, H", B, H"; end if;
end proc;
```

### \*Nouvelle fonction de cycle spécifique à l'étape 3:

Voici la bête, dommage qu'elle ne marche qu'ici...

```
Etape3EstCycle :=proc(Rubix, a,$)
    local Rubix2;
    Rubix2 := Rafraichir(RUBIX.FINI);
    if Rubix[a][coul][1] = Rubix2[a][coul][1] and Rubix[a][coul][2]
    = Rubix2[a][coul][2] and Rubix[a][coul][3] = Rubix2[a][coul][3] then 0;
    elif Rubix[a][coul][1] = Rubix2[a][coul][2] and Rubix[a][coul][2]
    = Rubix2[a][coul][3] and Rubix[a][coul][3] = Rubix2[a][coul][1] then 1;
    elif Rubix[a][coul][1] = Rubix2[a][coul][3] and Rubix[a][coul][2]
```

```
= Rubix2[a][coul][1] and Rubix[a][coul][3] = Rubix2[a][coul][2] then 2; end if;
end proc;
```

#### \*Fonction de scan:

Fonction intermédiaire de test.

```
Etape3Scan := proc(Rubix1,$)
local Rubix;
Rubix := Rafraichir(Rubix1);
Etape3EstCycle(Rubix, 3), Etape3EstCycle(Rubix, 4),
Etape3EstCycle(Rubix, 8), Etape3EstCycle(Rubix, 7);
end proc;
```

#### \*Fonction annexes d'éliminations:

Ces fonctions sont les mouvements à effectuer dans chacun des cas de figure, en fonction des orientations.

```
T21 := proc(a :: posint,$)
if a = 4 then Etape3Mouvement
elif a = 8 then H'', B, Etape3Mouvement, H, B''
elif a = 7 then H, B'', H, B'', Etape3Mouvement, H, B'', H, B''
elif a = 3 then H, B'', Etape3Mouvement, H'', B end if;
end proc;

T12 := proc(a :: posint,$)
if a = 4 then A, P'', H, B'', H, B'', Etape3Mouvement, H, B'', H, B'', A'', P
elif a = 8 then H'', B, A, P'', H, B'', H, B'', Etape3Mouvement, H, B'', H, B'', A'', P, H, B''
elif a = 7 then H, B'', H, B'', A, P'', H, B'', H, B'', Etape3Mouvement, H, B'', H, B'', A'', P,
H, B'', H, B''
elif a = 3 then H, B'', A, P'', H, B'', H, B'', Etape3Mouvement, H, B'', H, B'', A'', P, H'', B
end if;
end proc;

T111 := T12(4), T21(8) :
```

#### Fonction annexes d'éliminations:

Il s'agit de déplacements pour pouvoir appliquer les formules précédentes.

```
EC := proc(v, a,$)
if v[1] = a then v, [H, B'', B, H'']
elif v[2] = a then [v[2], v[3], v[4], v[1]], [H'', B]
elif v[3] = a then [v[3], v[4], v[1], v[2]], [H, B'', H, B'']
else [v[4], v[1], v[2], v[3]], [H, B''] end if;
end proc;

EC111 := v → EC(v, 0) :
EC210 := v → EC(v, 2) :
EC21 := v → EC(v, 2) :
```

#### Fonction annexes d'éliminations:

L'ultime étape de la résolution.

```
Etape3 := proc(Rubix,$)
local Rubix2, mouv, mouv2, s;
Rubix2 := Rafraichir(Rubix); mouv := NULL; mouv2 := NULL;
mouv2 := Etape3ChangementDeRef(Rubix2);
Rubix2 := Rafraichir(BougerRubix(Rubix2, mouv2, patron = false, volume = false,
cube = true));
s := Etape3Scan(Rubix2);
```

# Toujours en fonction des cas, cette fois plus complexes qu'à l'accoutumée.

```

if {s} = {0, 1} then mouv := ((EC111([s]))[2])[], T111;
elif {s} = {0, 2} then mouv := ((EC111([s]))[2])[], T111, T111;
elif {s} = {1, 2} then s := EC21([s]); mouv := (s[2])[];
    if [s[1][2], s[1][4]] = [1, 1] then mouv := mouv, T21(3),
T21(8);
    elif [s[1][2], s[1][4]] = [2, 1] then mouv := mouv, T21(4),
T12(7);
    elif [s[1][2], s[1][4]] = [1, 2] then mouv := mouv, T21(3),
T12(8); end if;
elif {s} = {0, 1, 2} then s := EC210([s]); mouv := ([s][2])[];
    if [s[1][2], s[1][4]] = [1, 0] then mouv := mouv, T21(3);
    elif [s[1][2], s[1][4]] = [0, 1] then mouv := mouv, T12(7);
    elif [s[1][2], s[1][4]] = [0, 0] then mouv := mouv, T21(3),
T21(4); end if;
end if;
mouv2, mouv,
end proc:

```

## D) Résolution

### Fonction de changement de référentiel:

Cette fonction simplifie une série de mouvements en prenant compte ceux qui correspondent à un "changement de référentiel" i.e. pivot sans mouvement de l'utilisateur.

*ChangementDeReferentiel :=proc()*

```

local s1, s, s2, j, k, A2, A2'', P2, P2'', R2, R2'', L2, L2'', H2, H2'',
B2, B2'';
    s1 := NULL;
    s2 := args;
    while nops([s2]) ≥ 2 do
        s := s21..2;
        if {s} = {H, B''} then
            A2 := L; A2'' := L'';
            P2 := R; P2'' := R'';
            R2 := A; R2'' := A'';
            L2 := P; L2'' := P'';
            H2 := H; H2'' := H'';
            B2 := B; B2'' := B'';
            s2 := s23..nops([s2]);
        elif {s} = {H'', B} then
            A2 := R; A2'' := R'';
            P2 := L; P2'' := L'';
            R2 := P; R2'' := P'';
            L2 := A; L2'' := A'';
            H2 := H; H2'' := H'';
            B2 := B; B2'' := B'';
            s2 := s23..nops([s2]);
        elif {s} = {R'', L} then
            A2 := B; A2'' := B'';
            P2 := H; P2'' := H'';

```

```

 $R2 := R; R2'' := R'';$ 
 $L2 := L; L2'' := L'';$ 
 $H2 := A; H2'' := A'';$ 
 $B2 := P; B2'' := P'';$ 
 $s2 := s2_{3..nops([s2])};$ 
elif {s} = {R, L''} then
     $A2 := H; A2'' := H'';$ 
     $P2 := B; P2'' := B'';$ 
     $R2 := R; R2'' := R'';$ 
     $L2 := L; L2'' := L'';$ 
     $H2 := P; H2'' := P'';$ 
     $B2 := A; B2'' := A'';$ 
     $s2 := s2_{3..nops([s2])};$ 
elif {s} = {A'', P} then
     $A2 := A; A2'' := A'';$ 
     $P2 := P; P2'' := P'';$ 
     $R2 := H; R2'' := H'';$ 
     $L2 := B; L2'' := B'';$ 
     $H2 := L; H2'' := L'';$ 
     $B2 := R; B2'' := R'';$ 
     $s2 := s2_{3..nops([s2])};$ 
elif {s} = {A, P''} then
     $A2 := A; A2'' := A'';$ 
     $P2 := P; P2'' := P'';$ 
     $R2 := B; R2'' := B'';$ 
     $L2 := H; L2'' := H'';$ 
     $H2 := R; H2'' := R'';$ 
     $B2 := L; B2'' := L'';$ 
     $s2 := s2_{3..nops([s2])};$ 
else  $A2 := A; A2'' := A'';$ 
       $P2 := P; P2'' := P'';$ 
       $R2 := R; R2'' := R'';$ 
       $L2 := L; L2'' := L'';$ 
       $H2 := H; H2'' := H'';$ 
       $B2 := B; B2'' := B'';$ 
       $s1 := s1, s2_1;$ 
       $s2 := s2_{2..nops([s2])};$ 
end if;
 $s := \text{NULL};$ 
for  $j$  in  $s2$  do
    if  $j = A$  then  $s := s, A2;$ 
    elif  $j = A''$  then  $s := s, A2'';$ 
    elif  $j = P$  then  $s := s, P2;$ 
    elif  $j = P''$  then  $s := s, P2'';$ 
    elif  $j = R$  then  $s := s, R2;$ 
    elif  $j = R''$  then  $s := s, R2'';$ 
    elif  $j = L$  then  $s := s, L2;$ 
    elif  $j = L''$  then  $s := s, L2'';$ 
    elif  $j = H$  then  $s := s, H2;$ 
    elif  $j = H''$  then  $s := s, H2'';$ 

```

```

elif  $j = B$  then  $s := s, B2;$ 
elif  $j = B''$  then  $s := s, B2'';$ 
    end if; end do;
 $s2 := s;$ 
end do;
 $s1, s2;$ 
end proc:

```

### Fonction de simplification:

Cette fonction simplifie certains mouvements débiles comme A,A'' ou A,A,A

```

Simplifier :=proc( )
    local mouv, k, l;
     $k := 1;$ 
    mouv := [args];
    l := nops(mouv);

    for k while  $k \leq l - 2$  do
        if {mouv[k], mouv[k + 1], mouv[k + 2]} = {A} then mouv := [op(1..k - 1,
mouv), A'', op(k + 3..l, mouv)]; l := nops(mouv); end if;
            if {mouv[k], mouv[k + 1], mouv[k + 2]} = {A''} then mouv := [op(1..k - 1,
mouv), A, op(k + 3..l, mouv)]; l := nops(mouv); end if;
                if {mouv[k], mouv[k + 1], mouv[k + 2]} = {P} then mouv := [op(1..k - 1,
mouv), P'', op(k + 3..l, mouv)]; l := nops(mouv); end if;
                    if {mouv[k], mouv[k + 1], mouv[k + 2]} = {P''} then mouv := [op(1..k - 1,
mouv), P, op(k + 3..l, mouv)]; l := nops(mouv); end if;
                        if {mouv[k], mouv[k + 1], mouv[k + 2]} = {H} then mouv := [op(1..k - 1,
mouv), H'', op(k + 3..l, mouv)]; l := nops(mouv); end if;
                            if {mouv[k], mouv[k + 1], mouv[k + 2]} = {H''} then mouv := [op(1..k - 1,
mouv), H, op(k + 3..l, mouv)]; l := nops(mouv); end if;
                                if {mouv[k], mouv[k + 1], mouv[k + 2]} = {B} then mouv := [op(1..k - 1,
mouv), B'', op(k + 3..l, mouv)]; l := nops(mouv); end if;
                                    if {mouv[k], mouv[k + 1], mouv[k + 2]} = {B''} then mouv := [op(1..k - 1,
mouv), B, op(k + 3..l, mouv)]; l := nops(mouv); end if;
                                        if {mouv[k], mouv[k + 1], mouv[k + 2]} = {R} then mouv := [op(1..k - 1,
mouv), R'', op(k + 3..l, mouv)]; l := nops(mouv); end if;
                                            if {mouv[k], mouv[k + 1], mouv[k + 2]} = {R''} then mouv := [op(1..k - 1,
mouv), R, op(k + 3..l, mouv)]; l := nops(mouv); end if;
                                                if {mouv[k], mouv[k + 1], mouv[k + 2]} = {L} then mouv := [op(1..k - 1,
mouv), L'', op(k + 3..l, mouv)]; l := nops(mouv); end if;
                                                    if {mouv[k], mouv[k + 1], mouv[k + 2]} = {L''} then mouv := [op(1..k - 1,
mouv), L, op(k + 3..l, mouv)]; l := nops(mouv); end if;
                                                        end do;

                                                        for k while  $k \leq l - 1$  do
                                                            if {mouv[k], mouv[k + 1]} = {A, A''} or {mouv[k], mouv[k + 1]} = {P, P''}
or {mouv[k], mouv[k + 1]} = {B, B''} or {mouv[k], mouv[k + 1]} = {H, H''}
or {mouv[k], mouv[k + 1]} = {R, R''} or {mouv[k], mouv[k + 1]} = {L, L''} then mouv
:= [op(1..k - 1, mouv), op(k + 2..l, mouv)]; l := nops(mouv); end if;
end do;

op(1..l, mouv);
end proc:

```

### \*Fonction de résolution d'un Rubik's cube:

Et voici enfin la tant attendue fonction de résolution du Rubik's cube, et avec toute sa panoplie d'options

```
ResoudreIII :=proc(Rubix1, {nbcoups :: boolean := true, commandes :: boolean := false, v
:: integer := 2, nbimage :: integer := 3, vol :: boolean := true, pat :: boolean := false},
$)

# nbcoup indique si l'on revoie le nombre de coups pour résoudre le Rubik's cube

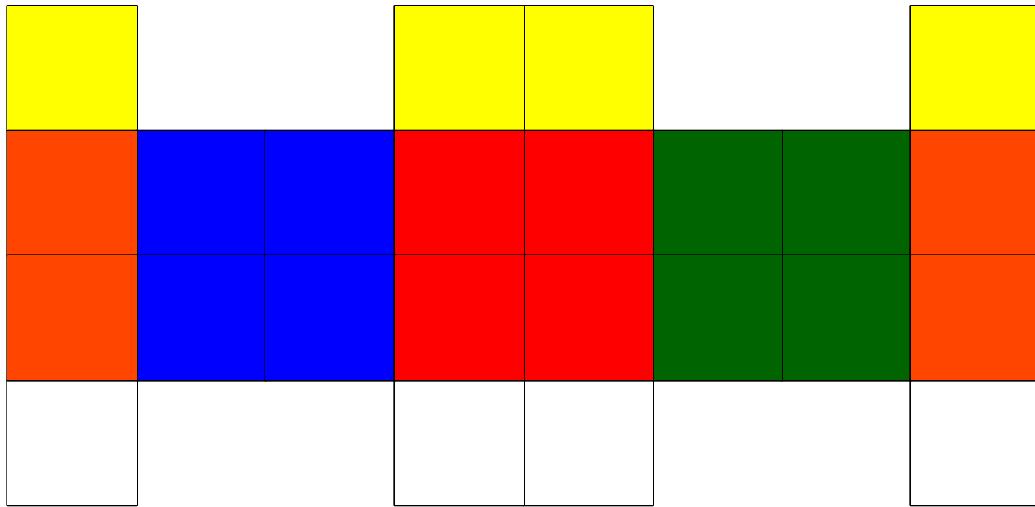
# commandes indique si l'on renvoie la liste des mouvements à effectuer pour la résolution
# nbimage indique le nombre d'images à afficher pour 1 mouvement
# vol indique si l'on affiche le volume des mouvements de la résolution
# pat indique si l'on affiche le patron des mouvements de la résolution
local Rubix, Mouv, Mouvements;
Rubix := Rafraichir(Rubix1) :
# On s'arrête si le cube est irrésoluble
if not EstFaisable(Rubix) then print("Rubik's cube irrésoluble"); break;
end if,
# Sinon on effectue successivement chaque étape
Mouv := Etape1(Rubix) : Mouvements := Mouv :
Rubix := Rafraichir(BougerRubix(Rubix, Mouv, patron =false, volume
=false, cube =true)) :
Mouv := Etape2(Rubix) : Mouvements := Mouvements, Mouv :
Rubix := Rafraichir(BougerRubix(Rubix, Mouv, patron =false, volume
=false, cube =true)) :
Mouvements := Mouvements, Etape3(Rubix) :
Mouv := Mouvements : Mouvements
:= ChangementDeReferentiel(Simplifier(Mouv)) :
# On simplifie de manière optimale les mouvements
while nops([Mouv]) ≠ nops([Mouvements]) do
Mouv := Mouvements : Mouvements
:= ChangementDeReferentiel(Simplifier(Mouv)) :
end do;
#Et on renvoie le tout
if nbcoups then printf("Ce Rubik's cube a été résolu en %g coups",
nops([Mouvements])); end if;
if vol or pat then BougerRubix(Rubix1, Mouvements, patron =pat, volume
=vol, cube =false, image =nbimage); end if;
if commandes then Mouvements; end if;
end proc;
```

## Utilisation

Pour représenter le patron d'un rubick's cube donné on fait:

```
DessinePatron(RUBIX.FINI);
```

## Patron du cube

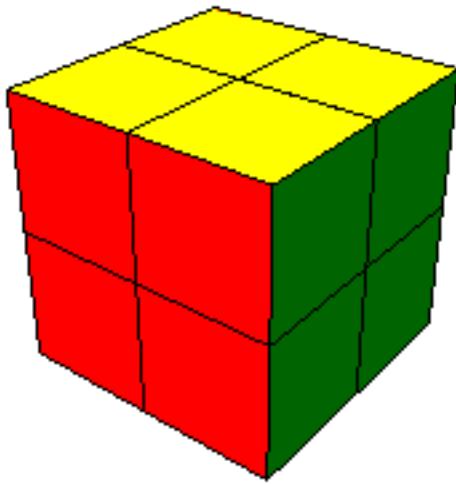


Pour représenter le volume d'un rubick's cube donné on fait:

!!! Attention: cette fonction est principalement utilisée pour afficher les mouvements, aussi l'argument est un vecteur de 3 coordonnées: le rubick's et deux autres arguments inutiles ici (théoriquement un angle et une face)

*DessineVolume( [RUBIX.FINI, 0, 0]);*

## Représentation 3D du cube

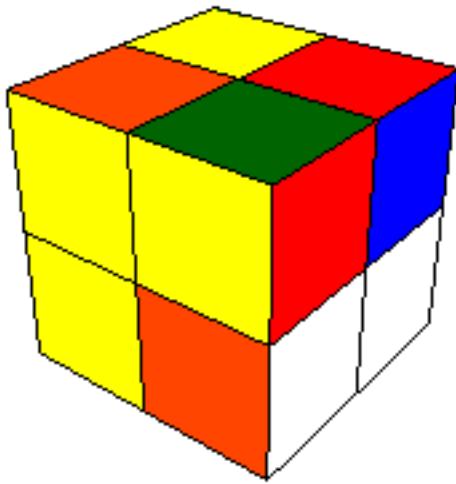


Pour créer un rubick's cube, on peut le faire soit aléatoirement:

*Rubix1 := CreerRubixAleatoire( ) :*

*DessineVolume( [Rubix1, 0, 0]);*

## Représentation 3D du cube

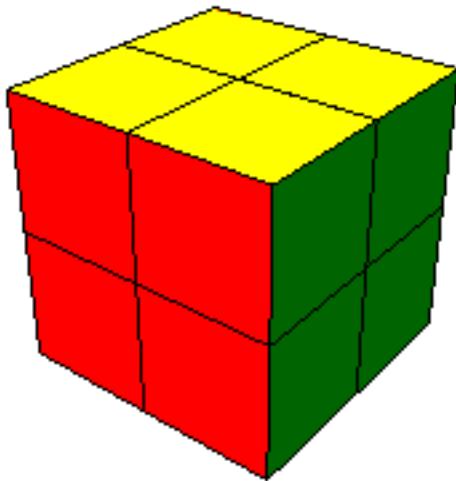


Ou alors en entrant les 24 couleurs dans l'ordre:

(avec rouge = r, vert = g, blanc = w, jaune = y, orange = o, bleu = b)

*Rubix2 := CreerRubix(r, r, r, r, g, g, g, g, o, o, o, o, b, b, b, b, w, w, w, w, y, y, y, y) :  
DessineVolume( [Rubix2, 0, 0]);*

## Représentation 3D du cube



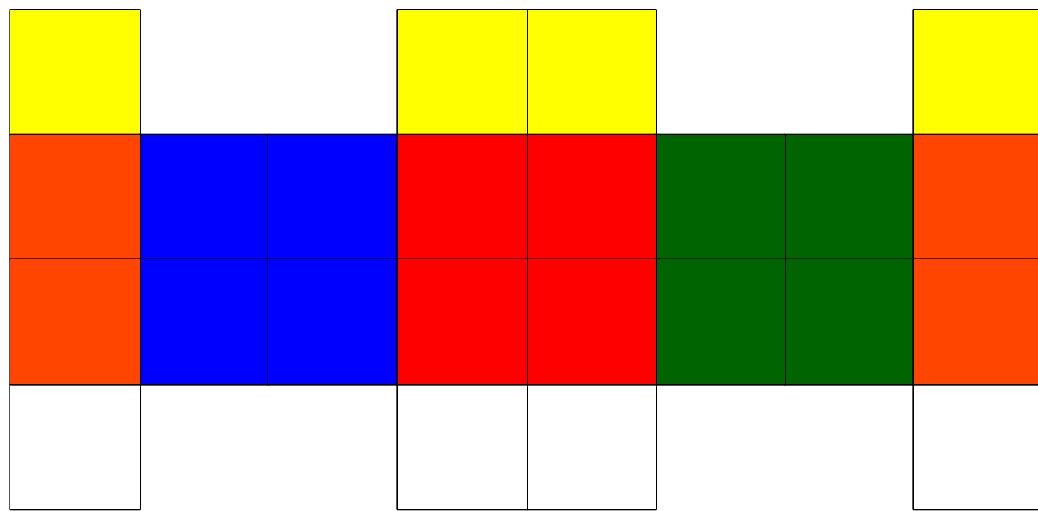
Pour bouger le rubick's cube, on procède comme suit:

(A = face avant sens direct, A" = face avant sens indirect, et de même P = posterieur, R = droite, L = gauche, H = haut, B = bas)

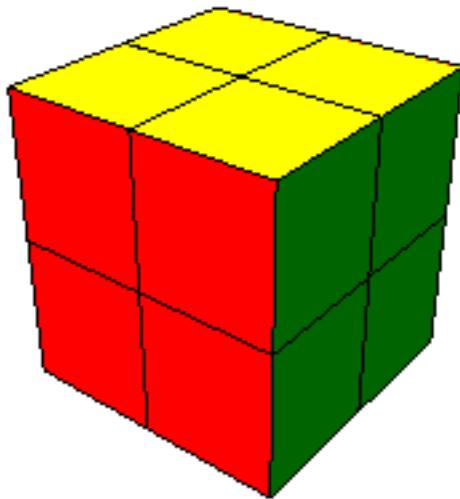
(les options ne sont pas nécessaires mais peuvent être choisies: patron (afficher le patron), volume (afficher le volume), cube (renvoyer le rubick's cube pivoté), image (nombre d'image par mouvement))

*BougerRubix(Rubix2, A, H, A, H, volume = true, patron = true, cube = false, image = 5);*

Patron du cube



## Représentation 3D du cube



Enfin pour résoudre un rubick's cube donné, on peut choisir la méthode de résolution I ou III (la II n'est pas disponible dans cette version):

(les options ne sont pas nécessaires mais peuvent être choisies: pat (afficher le patron), vol (afficher le volume), nbimage (nombre d'image par mouvement), nbcoups (affichage du nombre de coups nécessaire pour la résolution), commandes (affiche la liste des mouvements à effectuer pour la résolution))

!!! Attention: il est très fortement déconseiller d'afficher le volume pour la 1ère méthode sans nbimage=1 car avec 1000 coups en moyenne, la durée d'affichage est simplement monstrueuse.

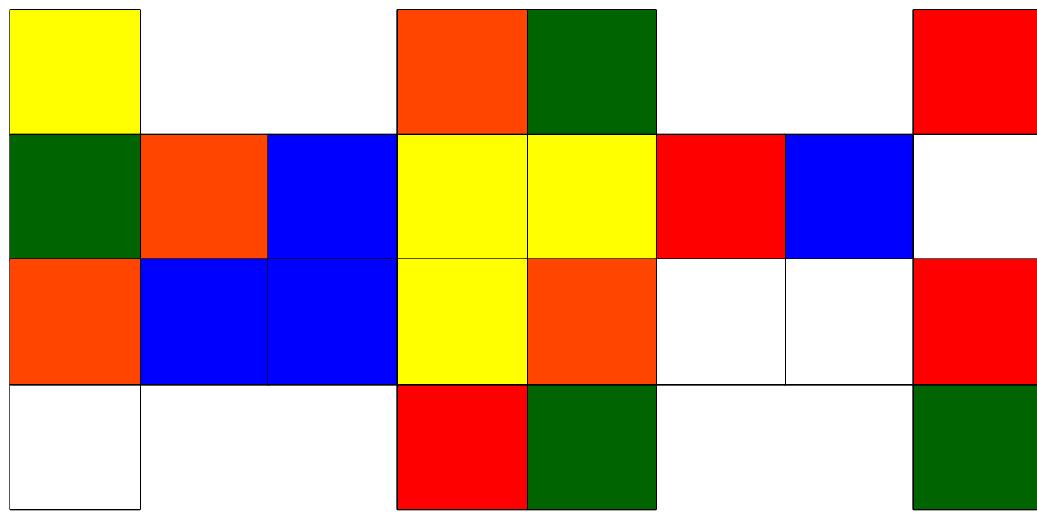
*ResoudreI(Rubix1, vol = false, pat = false, nbimage = 1, nbcoups = true, commandes = false);*

Cube résolu en 870 coups

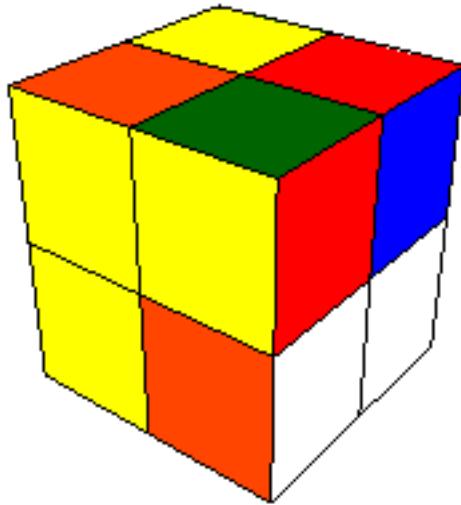
*ResoudreIII(Rubix1, vol = true, pat = true, nbimage = 3, nbcoups = true, commandes = true);*

Ce Rubik's cube a été résolu en 36 coups

Patron du cube



Représentation 3D du cube



*RotationBlocA, RotationBlocR", RotationBlocR", RotationBlocA", RotationBlocA",  
RotationBlocH", RotationBlocA, RotationBlocH, RotationBlocH, RotationBlocL,  
RotationBlocH", RotationBlocL", RotationBlocH", RotationBlocL", RotationBlocH,  
RotationBlocR, RotationBlocH", RotationBlocL, RotationBlocH, RotationBlocR",  
RotationBlocH, RotationBlocB, RotationBlocH, RotationBlocR", RotationBlocH",  
RotationBlocA", RotationBlocR", RotationBlocA, RotationBlocL", RotationBlocA",  
RotationBlocR, RotationBlocA, RotationBlocH, RotationBlocR, RotationBlocH",  
RotationBlocL* (6.1)