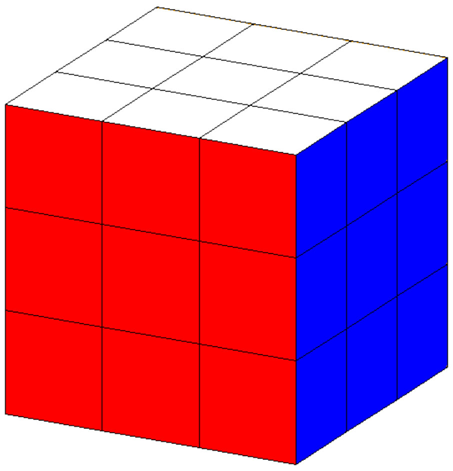
# Building and Solving Rubik's Cube in Mathworks® Matlab®.

Joren Heit

September 26, 2011



# Contents

## 1 Introduction

The Rubik's Cube was invented by the Hungarian architect Ernö Rubik in 1974 and was then still called "the Magic Cube". It was renamed after its inventor in 1980, after which it became immensely popular. So much even so, that it is hard to find people that have never tried to solve it. Although it is a toy that can be sold to children in toy shops, it has been the subject of many mathematical inquiries and it was not until July 2010 that it greatest secret was revealed. The simple mechanism on which the cube relies, can produce over 40 quintillion different states (1 quintillion = $10^{18}$) and this meant that up until 2010, not even computers were capable of finding out how many moves would be sufficient to solve any state. The shortest solution to a given state is called *God's Algorithm*, and the upper-bound on the number of moves necessary to solve any given state is called *God's Number* which turned out to be 20 [4].

This project was initiated out of curiosity, not knowing any of the above, let alone anything below. I started modelling the cube with the intention to solve it using a genetic algorithm that had no information whatsoever. This seemed convenient since I myself had no knowledge about the subject either, and I was not planning on memorizing or programming sets of algorithms to solve the cube, like most beginners do. However, the genetic algorithm did not come very far because it relied on a so-called fitness function (a measure of how close the cube is to being solved) that only counted the number of correctly placed facelets, which just isn't enough. Up to this date, I haven't revisited this approach using the tools I have at hand now, but this might be something interesting for later.

What I *did* end up with after all those hours of programming, debugging and frustration, was a working model of the cube. Still very much wanting to see my computer solve it, I consulted a friend of mine who told me that it might be possible to program a Layer-by-Layer method that is normally used by beginners. This is exactly what I did and yes, my computer was able to solve any given cube now, albeit in a lot of moves, averaging at around 110. When I read about God's Number, all of a sudden the algorithm I had been so proud of seemed utterly useless and I started a websearch for something better.

What I found was a website by Jaap Scherphuis [1] on which he has posted a letter from the mathematician Morwen Thistlethwaite to David Singmaster, dating from 1981. In this letter, Thistlethwaite describes the method he devised to solve the cube within 52 moves when done manually. He also states that when a computer search is performed, this number can be reduced to 45 (with an average of around 31). Despite the fact that there are more efficient methods out there, like Kociemba's Two Phase Algorithm which relies on Thistlethwaite's 45 move algorithm, I decided that this was a good starting point.

Still being very new on the subject, and also on the subject of Group Theory, which is central in Thistlethwaite's solving mechanism, it took me quite some time to fully understand the method and then implement it. Luckily, Jaap Scherphuis, Herbert Kociemba and Morwen Thistlethwaite himself have been very helpful and I now dare to say that, thanks to them, I now know what I'm doing despite the fact that I still do not know how to solve the cube when I'm not alowed to turn on my computer.

This documentation is meant for anyone interested in what I did and how I did it, but its main purpose is to solidify all the effort I put into the project, using another medium besides the Matlab codes and scripts. It basically consists of all the knowledge one needs to understand the cube, a description of the models I wrote to simulate the cube, and an explanation of the solving mechanisms.

# 2 Cube Theory

Before the cube models and the solving mechanisms are explained, it might be helpful to have a brief look at some of the most essential aspects of cube theory. This will be limited to the $3 \times 3 \times 3$ case, but can easily be extended to other cases by the reader. Furthermore, there are sevral sources on the web that can be consulted (for example, reference [2]).

## 2.1 Definitions

The cube consists of 20 so-called subcubes or *cubies* that are attached to the cube centres. These can be either edge or corner cubies that have either two or three colored (visible) faces. A colored cubie-face is from now on called a facelet, whereas a total cube-face (consisting of 9 facelets) is simply called a face. Each face can be rotated around its central axis (through the middle facelet), and such a rotation is called a *move*. In this report, the half-turn-metric (HTM, also known as face-turn-metric or FTM) will be employed. This means that both a quarter-turn (90°, 270°) and a half-turn (180°) will be counted as 1 move. The quarter-turn-metric (QTM) would have counted each half-turn as two moves. In the HTM, a total number of 18 different moves is then possible since each face can be rotated in 3 different ways. The moves are named after the face that is rotated:

- `L(eft)`
- `R(ight)`
- `F(ront)`
- `B(ack)`
- `U(p)`
- `D(own)`
- `I(dentitity operator)`

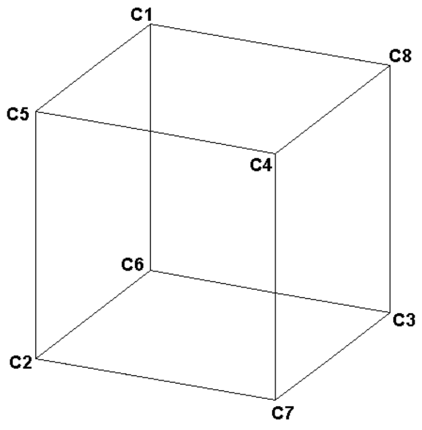
Each of these letters indicates a clockwise 90° rotation of the corresponding face (when looking directly at it). The postfixes '`'`' and '2' are added to specify a counterclockwise rotation and a 180° (half) turn respectively. For example, the move sequence `F2,B',U` would tell you to rotate the front-face 180°, followed by a counterclockwise 90° turn of the back-face, followed by a clockwise rotation of the up-face. The identity operator does not change the state of the cube, i.e. when an arbitrary move or sequence $X$ and its inverse $X^{-1}$ are applied right after eachother, this can be denoted as `I`.

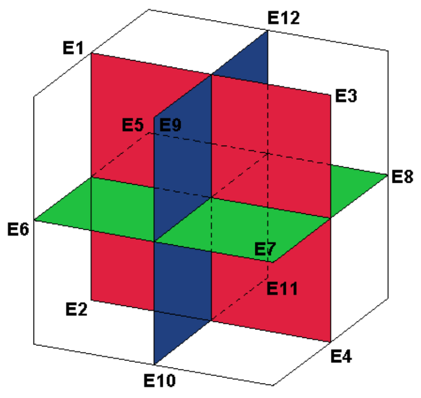
$$XX^{-1} = X^{-1}X = \mathtt{I} \tag{1}$$

## 2.2 Permutations

Now that the basic cube nomenclature is clear, we can focus on the *laws of the cube*, and compute the number of possible (legal) cube-states. By applying moves to the cube, the cubies get rearranged in a way that is called a permutation. However, the edge-cubies can never take a corner position and vice versa, so we could say that the edge pieces and corner pieces are permuted seperately of eachother. To denote these permutations, we could use various notations but first, we need to number the positions: corners 1-8 and edges 1-12. The order in which to number them is a matter of convention and may be chosen arbitrarily, but we will follow a convention by Morwen Thistlethwaite for reasons that will become clear when solving the cube (Section 5.3. This convention is illustrated in Fig. 1. The numbering might seem peculiar at first, but as can be seen in the figure, the edges are numbered per *slice* which are colored red, green and blue [1]. A slice is a set of 4 cubies (and 4 centres) that is within two other sets that each form a face. For instance, the `LR` slice denotes the slice that is within the `L` and `R` face (blue). The corners are numbered per *orbit*, which means the set of 4 positions that a corner piece can assume when only half-turns are allowed, i.e. when a corner piece starts in position C1-4, it won't be able to move to position C5-8 and vice versa.

Figure 1: Numbering of the corner and edge positions of the cube. The corner positions were numbered according to (half-turn) orbit, whereas the edge positions are numbered according to slice (`FB` (red),`LR` (blue),`UD` (green)).





A permutation of, for example, the corner pieces can now be represented as a $2 \times 8$ array where the first row holds the numbers 1-8 and the second holds the corner-numbers that actually are in these positions. For example, the corner-permutation of a solved cube in this notation would look like this:

$$p_{c,I} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{pmatrix} \tag{2}$$

One can immediately see that corner 1 is in position 1, corner 2 is in position 2, etc. This means that all the corner pieces are in the correct position. When the move `F` is applied to this permutation, corner 2 moves to position 5, corner

[1]Actually, Thistlethwaite did make an error in numbering the edges. The `FB` slice is numbered in a different way compared to the rest. He states in a letter to David Singmaster that this is due to a typing error in one of his programs and has stuck ever since. For debugging purposes, I decided to maintain this order anyway.

5 moves to position 4, etc. The resulting permutation is now expressed as

$$p_{c,F} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 1 & 7 & 3 & 5 & 2 & 6 & 4 & 8 \end{pmatrix} \tag{3}$$

This notation is called *two-line-notation* and can be simplified to *one-line-notation* by omitting the first row. A third, sometimes even more compact way of notation is called *cyclic notation* in which one denotes how the elements are cycled in order to achieve the permutation. For example, when corner 1 and two have switched positions, this is denoted as $(1, 2)$. If in addition to this, also corners 3 and 4 switched, the total permutation looks like $(1, 2)(3, 4)$. These were examples of *two-cycles* (or transpositions) but also cycles of higher order are possible. In the case of the `F` move we considered earlier, this would be denoted as:

$$p_{c,F} = (2745) \tag{4}$$

indicating that corner 2 moved to position 7, 7 to 4, 4 to 5 and 5 to 2.

There are two types of permutations: even and odd. An even permutation is one that can be expressed as a composition of an even number of transpositions (swaps). The corner-permutation $p_{c,F}$ is an odd permutation because it can only be expressed as a composition of 3 transpositions:

$$p_{c,F} = (2745) = (27)(25)(45) \tag{5}$$

The same holds for

$$p_{e,F} = (1, 2, 3, 4, 5, 10, 9, 8, 6, 7, 11, 12) = (6, 10)(7, 9)(6, 7) \tag{6}$$

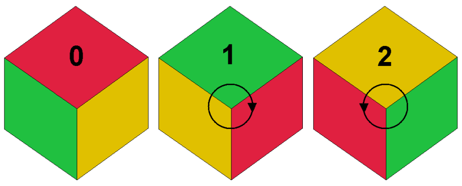
which combines to a total number of 6 transposition, which is even. It can easily be seen that this must hold for each move, and this leads to the first law:

**Cube Law 1.** *Only permutations of even parity can be reached.*

## 2.3 Orientations

Once the permutation of all the cubies is known, its state is not yet fuly determined. Each cubie possesses a second property: its orientation. An edge cubie can have either a good or a bad orientation and this is denoted with a 1 and a 0 respectively. For the corner-cubies, things are slightly more complicated since there are three possibilities for its orientation. When it is in the natural 'solved' position, this will be denoted by 0, and when it is rotated clockwise or counterclockwise the orientation will be denoted 1 or 2 (Fig. 2). We now need to define a frame of reference in which we can measure the orientation of each cubie. In the case of a corner cubie (which will always have a facelet belonging to either the `L` or `R` face), we'll say that a cubie has orientation 0 if the `L` or `R` facelet is on either the `L` or `R` face. Otherwise, its orientation will be determined according to the rule illustrated in Figure 2. In the case of an edge-cubie, we'll define an orientation as 'good' when the piece can be brought back to its original position in the right orientation without rotating the Up or Down faces (or using an even number of U/D-turns). The reason for this definition will become appearant when Thistlethwaite's algorithm is exlpained in Section 4.2. To denote a corner or edge orientation, the numbering from Figure 1 will

Figure 2: Rotation of a corner cubie around a diagonal axis, resulting in three different orientations: 0 (solved), 1 (clockwise), and 2 (counterclockwise).



be used. The corner- and edge-orientation, $o_c$ and $o_e$ can now be expressed as vectors of length 8 and 12 respectively.

Using these definitions, only a `U` or a `D` turn will flip the edge-orientation of the `U` or `D` cubies, whereas an `L` or `R` turn will preserve the corner-orientation of the `L` or `R` pieces. When we look at these properties more closely, we can extract the 2nd and 3rd laws of the cube. First we'll take a closer look at the edge-orientation, which can only change if either the `U` or `D` face is turned. When this happens, all 4 cubies on the turning face will flip their orientations. One might succeed in finding a move sequence that will eventually flip 2 cubies, but it is impossible to flip an odd number of cubie-orientations.

**Cube Law 2.** *The sum of all edge-orientations must be even, i.e.*

$$\sum_i o_{c,i} \mod 2 = 0 \tag{7}$$

The law that can be derived from the definition of corner-orientation is similar in the sense that a rotation of the `U,D,F,B` slices will rotate two cubies clockwise and the other two counterclockwise. This leads to the 3rd law.

**Cube Law 3.** *The sum of all corner-orientations must be a multiple of 3, i.e.*

$$\sum_i o_{e,i} \mod 3 = 0 \tag{8}$$

The total state can be summarized in a $2 \times 20$ matrix $\mathbf{S}$, holding all the cube's properties, and meeting the conditions set by the laws described above.

$$\mathbf{S} = \begin{pmatrix} c_p & e_p \\ c_o & e_o \end{pmatrix} \tag{9}$$

where the permutations are denoted in one-line-notation.

## 2.4 Cube Space

Without taking into account the restrictions imposed by the three laws of the cube, the cubies could be rearranged in a large number of ways. There would be $N_1 = 3^8$ ways of orienting the corner pieces and $N_2 = 8!$ ways to permute them. The edges could be oriented in $N_3 = 2^{12}$ different ways and arranged in $N_4 = 12!$ different permutations. This factors to a total number

$$N_{all} = N_1 \times N_2 \times N_3 \times N_4 \approx 5.2 \times 10^{20} \tag{10}$$

However, the three laws allow for only half of the permutations (first law), half of the edge orientations (second law) and one third of the corner orientations (third law). The total number of *reachable* states can then be calculated to be[2]

$$N = \frac{1}{12} N_{all} \approx 4.3 \times 10^{19} \tag{11}$$

## 3 The Cube Models

The tools and laws that were just discussed in Section 2 can now be used to construct our model of the cube. This will be done in two ways: on a facelet-level and on a cubie-level. The first will be convenient when the cube is plotted, whereas the second will be faster and more convenient in the solving phase. Let us start with the very intuitive facelet model.

### 3.1 Facelet Model

In the facelet model, the cube is represented by a multidimensional $d \times d \times 6$ array $\mathbf{R}$, holding the faclet-colors $R^k_{i,j} \in [1,6]$ of all 6 faces in 6 $d \times d$ matrices.

$$\mathbf{R^k} = \begin{pmatrix} R^k_{1,1} & \cdots & R^k_{1,d} \\ \vdots & \ddots & \vdots \\ R^k_{d,1} & \cdots & R^k_{d,d} \end{pmatrix} \tag{12}$$

We will now write a more general $d$ to specify the dimension, which up until now was assumed to be 3. We will go through this extra trouble because we want to be able to construct and plot and manipulate cubes of any size $d$. The order in which the matrices are stacked in the multidimensional array is now set to the following:

1. `F`
2. `R`
3. `B`
4. `L`
5. `U`
6. `D`

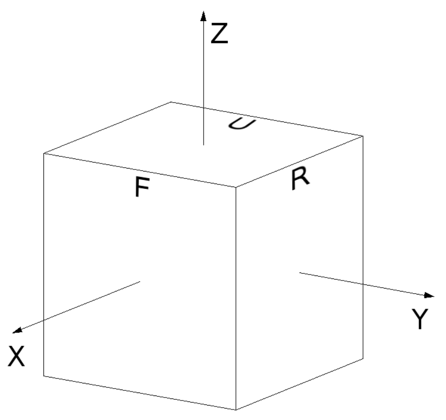
It was chosen this way because it would be the order if one would rotate the cube clockwise around the `UD` axis (end then look at the `U` and `D` faces themself).

With each move that is applied to the cube, $\mathbf{R}$ changes its state in a way that we will now work out. To do this, we will define the matrix-operators $^R$ and $^C$ in a way that they will inverse the order of the rows and columns of the matrix upon which they act, that is

$$A^R = \begin{pmatrix} A_{d,1} & \cdots & A_{d,d} \\ \vdots & \ddots & \vdots \\ A_{1,1} & \cdots & A_{1,d} \end{pmatrix} \tag{13a}$$

[2] $N = 43,252,003,274,489,856,000$ to be more precise

Figure 3: Reference frame used to define the generalised moves. Instead of using the traditional notation, we will use a more general way in which the rotation axis, row number and angle are specified. This allows for cubes of higher dimension to manipulated easily. A conversion table will be introduced later on which can be used to map the different notations onto one another.



$$A^C = \begin{pmatrix} A_{1,d} & \dots & A_{1,1} \\ \vdots & \ddots & \vdots \\ A_{d,d} & \dots & A_{d,1} \end{pmatrix} \tag{13b}$$

The operator $^T$ denotes the transpose of a matrix as usual. This allows us to write 90° clockwise and counterclockwise rotations of matrix-elements as

$$A^{90^\circ \circlearrowright} = (A^T)^C \tag{14a}$$

$$A^{90^\circ \circlearrowleft} = (A^T)^R \tag{14b}$$

$$A^{180^\circ \circlearrowright} = (A^R)^C = (A^C)^R \equiv A^{RC} \tag{14c}$$

When a face is turned, e.g. `F`, the corresponding matix in $\mathbf{R}$ is transformed using the oprations from equations 14 but this does not yet conclude the move. The four faces perpendicular to the one that is being rotated will exchange facelets and to specify this more accurately, we will need to define a frame of reference. This is illustrated in Figure 3, where the $x$-axis points out of the `F`-face, the $Y$-axis points out of the `R`-face and the $z$ axis points out of the `U`-face. We can now define 3 new matrices $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{z}$ that will each hold the 4 face-matrices $\mathbf{R_k}$ that are perpendicular to the direction they correspond to. One must be very cautious in determining the orientation of the matrices, for the elements must line up in the way they do on a real cube.

$$\mathbf{x} \equiv \left[ \ U \mid (R^T)^R \mid D^{RC} \mid (L^T)^C \ \right] \tag{15a}$$

$$\mathbf{y} \equiv \left[ \ (F^T)^C \mid (U^T)^C \mid (B^T)^C \mid (D^T)^C \ \right] \tag{15b}$$

$$\mathbf{z} \equiv \left[ \ F \mid R \mid B \mid L \ \right]^{\mathbf{C}} \tag{15c}$$

A move now needs to consist of 3 elements: the rotations axis ($r = x, y, z$), the row that is rotated ($i = 1, 2, \dots, d$) and the angle or number of rotations

$(k = 1, 2, 3(-1))$. To rotate the cube, we will just take the matrix from equation 15 specified by $r$, and cycle the $i^{\text{th}}$ row over a distance $d$, repeating this $k$ times. For example, when the move $rik = z11$ is called, the following transformation would take place:

$$\begin{aligned}
\mathbf{z} \ = & \left[\begin{array}{ccc|ccc|}
L_{1,d} & \dots & L_{1,1} & B_{1,d} & \dots & B_{1,1} \\
\vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
L_{d,d} & \dots & L_{d,1} & B_{d,d} & \dots & B_{d,1}
\end{array}\right. \\
& \left.\begin{array}{ccc|ccc}
R_{1,d} & \dots & R_{1,1} & F_{1,d} & \dots & F_{1,1} \\
\vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
R_{d,d} & \dots & R_{d,1} & F_{d,d} & \dots & F_{d,1}
\end{array}\right] \rightarrow
\end{aligned} \tag{16a}$$

$$\begin{aligned}
\mathbf{z}' = & \left[\begin{array}{ccc|ccc|}
F_{1,d} & \dots & F_{1,1} & L_{1,d} & \dots & L_{1,1} \\
\vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
L_{d,d} & \dots & L_{d,1} & B_{d,d} & \dots & B_{d,1}
\end{array}\right. \\
& \left.\begin{array}{ccc|ccc}
B_{1,d} & \dots & B_{1,1} & R_{1,d} & \dots & R_{1,1} \\
\vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
R_{d,d} & \dots & R_{d,1} & F_{d,d} & \dots & F_{d,1}
\end{array}\right]
\end{aligned}$$

$$\left[\begin{array}{c|c|c|c} F & R & B & L \end{array}\right]^C \leftarrow \mathbf{z}' \tag{16b}$$

When a double move or counterclockwise move was requested, these steps would be repeated 2 or 3 times, resulting in the desired endstate.

## 3.2 Cubie Model

A less intuitive but more efficient and compact way to model our cube is by using the tools we aquired in Section 2. Our cube can simply be represented by Equation 9 and each move is denoted as a permutation that acts on the cubies, plus a transformation of the orientations according to a simple set of rules. We are however, somewhat more limited to the $3 \times 3 \times 3$ case and while this is trivially reduced to the $2 \times 2 \times 2$ case (where the cube only has corners), it is harder for higher dimensions. We will stick to the standard $3 \times 3 \times 3$ cube in this description.

As mentioned earlier, each move permutes both the corner and edge cubies in very specific ways. The reader may check that, in one-line-notation, all the moves can be denoted like Equations 17, where we split them into two parts: $M_c^p$ being the effect on the corner-cubies and $M_e^p$ the effect on the edge-cubies.

$$\begin{aligned} F_c^p &= (1, 7, 3, 5, 2, 6, 4, 8) \\ F_e^p &= (1, 2, 3, 4, 5, 10, 9, 8, 6, 7, 11, 12) \end{aligned} \tag{17a}$$

$$\begin{aligned} B_c^p &= (6, 5, 3, 4, 1, 2, 7, 8) \\ B_e^p &= (1, 2, 3, 4, 12, 6, 7, 11, 9, 10, 5, 8) \end{aligned} \tag{17b}$$

$$\begin{aligned} L_c^p &= (6, 5, 3, 4, 1, 2, 7, 8) \\ L_e^p &= (5, 6, 3, 4, 2, 1, 7, 8, 9, 10, 11, 12) \end{aligned} \tag{17c}$$

$$\begin{aligned} R_c^p &= (1,2,8,7,5,6,3,4) \\ R_e^p &= (1,2,7,8,5,6,4,3,9,10,11,12) \end{aligned} \tag{17d}$$

$$\begin{aligned} U_c^p &= (5,2,3,8,4,6,7,1) \\ U_e^p &= (9,2,12,4,5,6,7,8,3,10,11,1) \end{aligned} \tag{17e}$$

$$\begin{aligned} D_c^p &= (1,6,7,4,5,3,2,8) \\ D_e^p &= (1,11,3,10,5,6,7,8,9,2,4,12) \end{aligned} \tag{17f}$$

Each of the above moves can now be applied to the cube to find the new corner- and edge-permutation:

$$p_c \to M_c^p(p_c) \tag{18a}$$

$$p_e \to M_e^p(p_e) \tag{18b}$$

When the new permutation is calculated, one must determine the new orientations of the pieces. By definition (Section 2.3) the orientation of the corner pieces is invariant under rotations of the `L` or `R` face, and the orientation of the edge pieces will only change when either the `U` or `D` face is twisted. Closer inspection of these properties will lead to the conclusion that the change of orientation can be expressed by adding a vector $M^o$ to the orientation-vectors $o_c$ and $o_e$. Equations 19 list these vectors.

$$\begin{aligned} F_c^o &= (0,1,0,1,-1,0,-1,0) \\ F_e^o &= (0\ldots0) \end{aligned} \tag{19a}$$

$$\begin{aligned} B_c^o &= (1,0,1,0,0,-1,0,-1) \\ B_e^o &= (0\ldots0) \end{aligned} \tag{19b}$$

$$\begin{aligned} U_c^o &= (-1,0,0,-1,1,0,0,1) \\ U_e^o &= (1,0,1,0,0,0,0,0,1,0,0,1) \end{aligned} \tag{19c}$$

$$\begin{aligned} D_c^o &= (0,-1,-1,0,0,1,1,0) \\ D_e^o &= (0,1,0,1,0,0,0,0,0,1,1,0) \end{aligned} \tag{19d}$$

The new orientation of the cubies can now be aquired by adding these vectors to the original orientation representations.

$$o_c \to (o_c + M_c^o) \mod 3 \tag{20a}$$

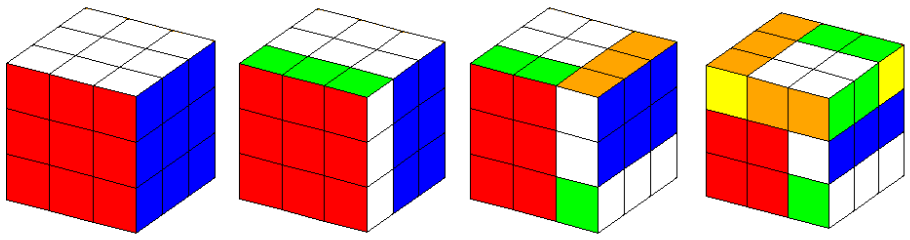
$$o_e \to (o_e + M_e^o) \mod 2 \tag{20b}$$

## 3.3 Plotting the Cube

Now, for the first time, we will employ some of the handy plotting features of Matlab. The cube-model as described above could have been simulated in any program or language, but when visualisation of the state is desired, Matlab offers some big advantages. We will make use of the facelet-model instead of the cubie model. When the state is only known as a set of permutations and orientations, these have to be converted to facelet colors, which can be done after some thought and effort. From now on, we will assume that the multidimensional array **R** is already constructed.

To plot the cube, we will make use of the `fill3()` function which is one of the standard plotting functions of Matlab. We will use it to draw a total number

Figure 4: Plots of a cube, using Matlab's `fill3()` function, while it gets manipulated by the sequence `F,R',U2`.



of 54 squares and fill it with the colors specified by **R**. To do this, it needs the coördinates of the corner vertices of each facelet, which depend on the dimension of the cube. Therefore, a $d \times d \times 6$ cell-array is constructed, with each cell holding the 4 coördinates of the corresponding facelet. The colormap is defined as below to assign the correct colors to each facelet and the cube can be plotted. To keep in touch with reality, the original Rubik's Cube color-convention is applied:

1. Red (`F`)
2. Blue (`R`)
3. Orange (`B`)
4. Green (`L`)
5. White (`U`)
6. Yellow (`D`)

Figure 4 displays what it now looks like when an unscrambled cube is manipulated by the sequence `F,R',U2`.

To increase the amount of eyecandy even more, an optional animation handle is built into our plotting function. When the command to animate a certain move is issued by the user, 5 intermediate steps will be plotted in order to create a movie-like animation. With the coördinates of each facelet already at hand, one only needs the appropriate rotation matrix to calculate the new coördinates after they have been rotated over an angle $\theta$ with respect to a certain axis. The three rotation matrices are listed below as Equations 21.

$$R_x(\theta) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{pmatrix} \tag{21a}$$

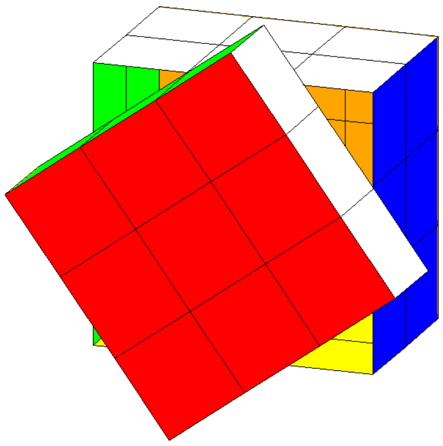
$$R_y(\theta) = \begin{pmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{pmatrix} \tag{21b}$$

$$R_z(\theta) = \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{21c}$$

When, for example, the move `F` needs to be animated, the coördinates of all `F`-cubies are calculated for $\theta = 18°, 36°, 54°, 72°, 90°$ as shown in Equation 22.

$$\begin{pmatrix} x_\theta \\ y_\theta \\ z_\theta \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{pmatrix} \begin{pmatrix} x_0 \\ y_0 \\ z_0 \end{pmatrix} = \begin{pmatrix} x_0 \\ y_0\cos(\theta) - z_0\sin(\theta) \\ y_0\sin(\theta) + z_0\cos(\theta) \end{pmatrix} \tag{22}$$

Figure 5: Snapshot from an animation when the `F`-face is being rotated clockwise. After 5 such steps, it arrives at the new state, which is then assigned to **R**.



The result is then plotted using `fill3()` and if the CPU is fast enough, this will result in a smooth transition from one state to another. A snapshot of such an animation is shown in Figure 5.

## 4 The Solving Algorithms

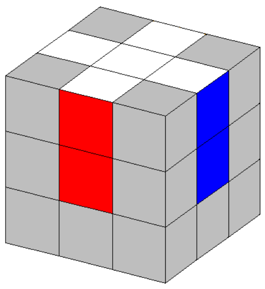
Now that the cube-toolbox is complete, it is time to start solving. Before one can solve a cube, it has to be scrambled. This basically means that sequence of random moves is applied to the cube without memorizing this sequence. This can easily done by our programs by simply generating a solved cube, generating a list of random moves using pseudo-random numbers and the applying these moves to the solved cube. As mentioned earlier in the Introduction, two different mechanisms to solve the cube $3\times3\times3$ cube. The first is a human *Layer-by-Layer* method, and the second is the more advanced *Thistlethwaite 45 Move Algorithm* which can only be done by computers (unless you're very, very patient). The first will now be described stage by stage. An implementation of this method is straightforward, though time consuming.

### 4.1 Layer-by-Layer

The human Layer-by-Layer method is a beginners' method because it requires the solver only a small number of sequences to memorize. In our case, this means that we only have to *pre-program* a small number of sequences. With these algorithms at hand, the cube can be solved in 6 stages:

1. Form a cross
2. Fix permutation/orientation of the 1st layer corners
3. Solve the middle-layer.
4. Form a cross on the bottom layer.
5. Fix the permutation of the bottom layer cornerpieces.
6. Solve the remaining edgepieces

Figure 6: The result of the first stage, when the cross has been formed on the first layer. Facelets are colored gray to indicate that these could have any color.



### 4.1.1 Stage 1: L1-Edges

The first stage is somewhat trivial, and even for beginners no algorithms are provided. With a little intuition, one can always find a short number of moves to build a cross on top. The only tricky part may be that not only should a cross form on the top face, but the colors of their adjecent facelets should match the face-colors (as identified by the centre-facelets) of the adjecent faces. This simply means that both the permutation and orientation of the L1-edges must be solved. There are many ways to let a computer solve this problem. The easiest one can be described in 5 steps:

1. Check all faces to see if a cross already exists.
2. If a cross exists on face $i$, re-orient the cube to make this face the `U`-face, then go to step 5. Otherwise pick the face that comes closest to having a cross and re-orient.
3. Iterate over the perpendicular faces (`F,R,B,L`) by rotating the cube and search for the remaining `U`-facelets.
4. When one is found, move this to an 'empty' spot on top, not caring about the other edge-facelet yet.
5. When a cross has formed, check which pieces should be exchanged to match the adjecent facelets to their face (fix permutation) and swap them around until the cross is completed. The trivial algorithms that can be employed are

   $$A_1 = \texttt{F2,B2,U2,F2,B2}$$

   to swap the front- and back-edge or

   $$A_2 = \texttt{F2,D,R2,D',F2}$$

   $$A_3 = \texttt{F2,D',L2,D,F2}$$

   to interchange the front- and right/left-edge.

The result of Stage 1 is illustrated in Figure 6.

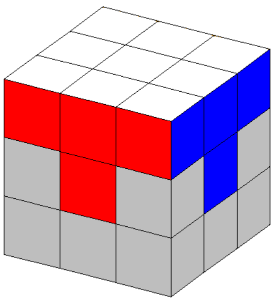
### 4.1.2 Stage 2: L1-Corners

Now that the L1-edges are all solved, the cornerpieces have to be solved as well to complete the first layer (which is assumed to be on top). To do this, we will need to move any cornerpieces that we can find to the `FRD` position, while the cube is oriented in a way that the `FUR` corner-position is occupied by a wrong cornerpiece. The piece can then be brought up by applying the algorithm

$$A_4 = \texttt{R',D',R,D}$$

iteratively until it is placed correctly. If one or more corners are occupied by wrongly placed L1-corners, these can be brought down to the bottom layer by applying the same algorithms. This method is simply repeated until all the cornerpieces are in their correct positions, as is shown in Figure 7.

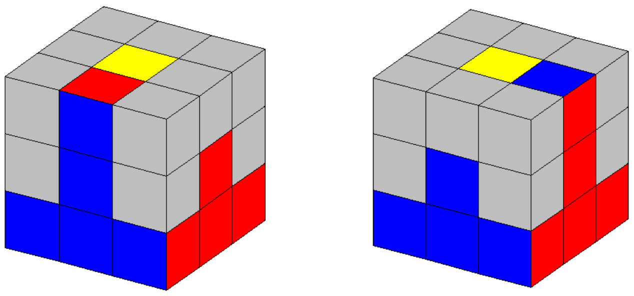
Figure 7: The result of the second stage, when the first layer has been solved completely.



### 4.1.3 Stage 3: L2-Edges

In this stage, the L2-edges are brought into place. For starters, the cube is re-oriented by making the solved layer (L1) face down, i.e. a 180° rotation of the entire cube around the $x$ or $y$ axis. The we search the top layer (L3) for edge-pieces that belong in L2 and we twist the `U`-face in order to place the piece to either the left or right of its final destination (Figure 8).

Figure 8: There are two ways to place the edge-piece in the appropriate position. When the situation is like that on the left and the piece has to be moved to the right, $A_5$ is used. When the right situation is at hand and the piece needs to be moved to the left, we have to use $A_6$ in order to solve it.



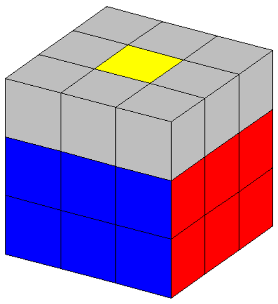
Both situations come with their own algortithms, which are mirror-images of one another:

$$A_5 = \texttt{U,R,U',R',U',F',U,F}$$

$$A_6 = \texttt{U',L',U,L,U,F,U',F'}$$

Where $A_5$ is used to move a piece to the right-edge and $A_6$ will move the piece to the left-edge. These algorithms can also be used to remove an edge piece from L2 and bring it back to L3 in order to re-position it and put it in the right spot using again one of these two algorithms. This is then repeated until L2 is completely solved, as shown in Figure 9.

Figure 9: Result after Stage 3, when both L1 and L2 are solved.



### 4.1.4 Stage 4: L3-Edge-Orientation

Again, we need to make a cross, this time on L3 which is still the top-layer. Despite the fact that we need not worry about the permutation yet, it is much more difficult to make a cross at this stage because we don't want to mess with the solved part of our cube. There are three unsolved possible situations at this point, plus ofcourse the situation where a cross has already formed (Figure 10,11).

1. No correctly oriented edges in L3 at all: the *dot-case*.
2. Two correctly oriented edges opposite of eachother: the *bar-case*.
3. Two correctly oriented edges adjecent to eachother: the *L-case*.

These situations correspond to the sum of the L3-edge-orientation being 0 (solved), 2 (bar/L) or 4 (dot). Any other situations are prohibited by the laws of the cube since these would violate the 2nd Law (Section 2.3).

Each of the three cases can be solved by using one of the following algorithms:

$$A_7 = \texttt{F,R,U,R',U',F'}$$

$$A_8 = \texttt{F,U,R,U',R',F'}$$

The solutions to each case are listed below:

| | | |
|---|---|---|
| 1. | (dot) | $A_7$, `U2`, $A_7$ |
| 2. | (bar) | Twist the `U`-face until the bar runs from left to right, then apply $A_7$. |
| 3. | (L) | Twist the `U`-face until the L-shape is in the `BL`-corner, then apply $A_8$. |

When done correctly, end-result should now look like the one in Figure 11.

Figure 10: Three possible situations at the start of Stage 4. From left to right, these are named the dot-, bar- and L-case.

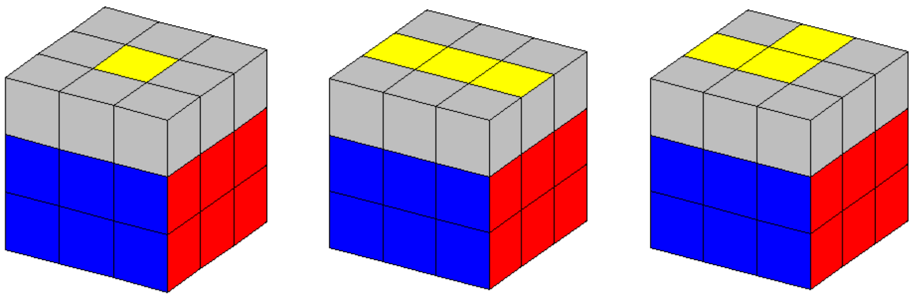
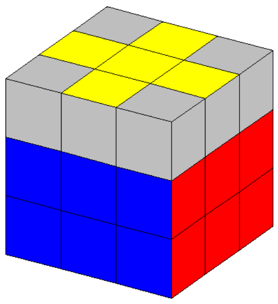


Figure 11: State of the cube after Stage 4 has been completed: all edges have the right orientation now.



### 4.1.5 Stage 5: L3-Corner-Permutation

In this stage, the corner-permutation of the L3-corners will be restored. The algorithm that is central in this stage is the following:

$$A_9 = \texttt{L,U',R',U,L',U',R,U2}$$

As a consequence of the 1st law, the corners can always be cycled (by turning the `U`-face) such that at least 2 corners are in the correct position. The other corners can then either adjecent to or diagonally opposite of eachother. In the first case, one has to orient the cube in such a way that these two corners are on the `R`-side and then apply $A_9$ to make them switch places. When the corners are diagonally opposite of eachother, $A_9$ is apllied to make the `FUR`-piece and `RUB`-piece trade places which brings us back to the first case. As a result, the corners are correctly permuted and can now be oriented in the next stage.

### 4.1.6 Stage 6: L3-Corner-Orientation

There are 8 orientation possibilities allowed by the 3rd law, which can all be solved by using one of, or a combination of, these algorithms:

$$A_{10} = \texttt{R',U',R,U',R',U2,R,U2}$$

$$A_{11} = \texttt{R,U,R',U,R,U2,R',U2}$$

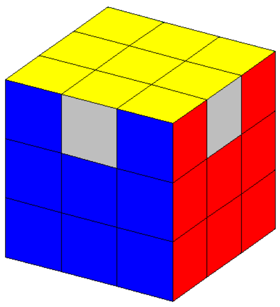
After applying $A_{10}$, all corners except for `LBU` are rotated counterclockwise whereas $A_{11}$ rotates all corners clockwise with the exception of `FLU`. This allows us to tabulate the different possible situations and their solutions in Table 1. Table 1 can now be used to fix all corner-orientations, after which the cube will

| | | | | |
|---|---|---|---|---|
| | 0 | 0 | | Solved |
| | 0 | 0 | | |
| | 0 | 1 | | $A_{10}$ |
| | 1 | 1 | | |
| | 2 | 2 | | $A_{11}$ |
| | 0 | 2 | | |
| | 2 | 0 | | $A_{11}, A_{10}$ |
| | 1 | 0 | | |
| | 2 | 1 | | `U2`, $A_{11}$, `U2`, $A_{10}$ |
| | 0 | 0 | | |
| | 2 | 0 | | `U`, $A_{11}$, `U'`, $A_{10}$ |
| | 0 | 1 | | |
| | 1 | 2 | | `U`, $A_{10}$, `U'`, $A_{10}$ |
| | 1 | 2 | | |
| | 2 | 1 | | `U2`, $A_{11}$, `U2`, $A_{11}$ |
| | 1 | 2 | | |

Table 1: This table lists the different possible corner-orientations and their solution. In order to solve this stage, one should re-orient the cube to one of the listed orientations (as seen from above) and perform the algorithm next to it.

look like Figure 12.

Figure 12: State of the cube after Stage 6 has been completed: all corners are correctly oriented now.



### 4.1.7 Stage 7: L3-Edge-Permutation

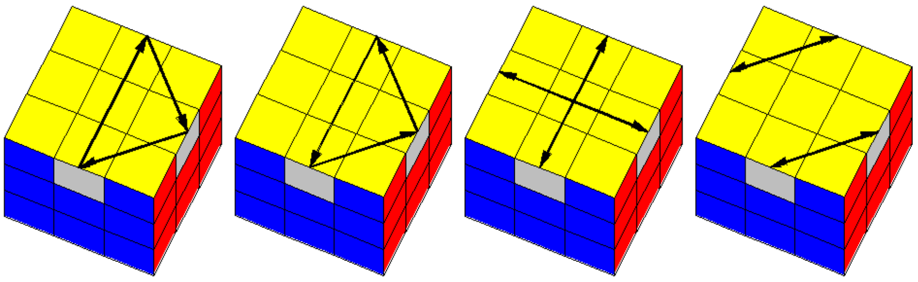
Due to the 1st law (Section 2.2), there are $^{4!}/_{2} = 12$ possible permutations that can occur now. This number can be reduced to 4 when considering the rotational symmetries. These situations are illustrated in Figure 13 and can be solved by using the final set of algorithms:

$$A_{12} = \texttt{R2,U,F,B',R2,F',B,U,R2}$$

$$A_{13} = \texttt{R2,U',F,B',R2,F',B,U',R2}$$

For the first two situations, $A_{12}$ and $A_{13}$ respectively are applied to the cube (when oriented as shown in Figure 13).

Figure 13: Schematic overview of the four possible edge-permutations at the start of stage 7. Arrows indicate how each edge-piece should move in order to get to its solved position.



For the last two situations, one may choose which of these algorithms to apply, bringing it back to one of the first two situations. The cube can now be re-oriented and solved using again one of the algorithms above. In more detail, each of the states in Figure 13 can be solved as follows:

1. $A_{12}$
2. $A_{13}$
3. $A_{12}$, clockwise 90° rotation around the $z$-axis, $A_{13}$
   $A_{13}$, counterclockwise 90° rotation around the $z$-axis, $A_{12}$
4. $A_{12}$, counterclockwise 90° rotation around the $z$-axis, $A_{12}$
   $A_{13}$, clockwise 90° rotation around the $z$-axis, $A_{13}$

The cube is now solved!

### 4.1.8 Recap: Commutators

Now that we established a mechanism to solve the cube, we must have a second look at the algorithms and find out why they work. This has to do with a phenomenon called commutators. A commutator is denoted as $[X, Y]$ where $X$ and $Y$ are arbitrary operators. In our case, they could be moves or even entire move sequences. The commutator $[X, Y]$ is then defined as

$$[X, Y] = XYX^{-1}Y^{-1} \tag{23}$$

where $X^{-1}$ denotes the inverse operation of $X$.

Imagine two operators that act on different parts of the cube, e.g. $X = \texttt{F}$ and $Y = \texttt{B}$. When $[X, Y] = \texttt{F,B,F',B'}$ is now performed on the cube, its state will not change because the moves cancel eachother out. This happens because there are no cubies on which both of the operations can act. When we define $S_X$ and $S_Y$ to be the sets of cubies on which the operators $X$ and $Y$ subsequently operate, there would in this example be no intersection between these two sets, i,e.

$$S_X \cap S_Y = S_{X^{-1}} \cap S_{Y^{-1}} = \emptyset \tag{24}$$

When there *is* an intersection between $S_X$ and $S_Y$ or $S_{X^{-1}}$ and $S_{Y^{-1}}$, the commutator will affect only the union of these intersections, and leave the other pieces unchanged.

$$S_{\text{aff}} = (S_X \cap S_Y) \cup (S_{X^{-1}} \cap S_{Y^{-1}}) \tag{25}$$

**Example 1.** The algorithm that was introduced in Stage 2 (Section 4.1.2) of the beginners' method is a nice example of a simple commutator:

$$A_4 = \texttt{R',D',R,D}$$

It can easily be seen that $X = \texttt{R'}$ and $Y = \texttt{D'}$. We can now find the two sets $S_X$ and $S_Y$, using the numbering conventions from Section 2.2 and keeping in mind that the first operator changes the state of the cube, thereby also the set of cubies that the second operator will work on. The notations $c_i$ and $e_i$ will denote the cubies that belongs to the positions $C_i$ and $E_i$.

$$S_X = \{c_3, c_4, c_7, c_8, e_3, e_4, e_7, e_8\} \tag{26a}$$

$$S_Y = \{c_2, c_4, c_6, c_7, e_2, e_7, e_{10}, e_{11}\} \tag{26b}$$

$$S_{X^{-1}} = \{c_3, c_6, c_7, c_8, e_3, e_4, e_8, e_1 1\} \tag{26c}$$

$$S_{Y^{-1}} = \{c_2, c_3, c_4, c_6, e_2, e_4, e_7, e_{10}\} \tag{26d}$$

The intersection of these sets can now be seen to be

$$(S_X \cap S_Y) \cup (S_{X^{-1}} \cap S_{Y^{-1}}) = \{c_3, c_4, c_6, c_7, e_4, e_7\} \tag{27}$$

The purpose of $A_4$ is to make the corners $c_4$ and $c_7$ trade places without disturbing the rest of the L1-cubies, which is exactly what we can make out of the intersection above, since $c_4$ is the only L1-cubie in the set from equation 27.

**Example 2.** One may have more difficulties in finding the commutators in $A_{12}$. Nevertheless, when we use the identities `F,B` = `B,F` and `U,U'` = `R2,R2` = `I`, we can rewrite the algorithm to a form where the commutators magically appear.

$$\begin{aligned} A_{12} &= \texttt{R2,U,F,B',R2,F',B,U,R2} \\ &= \texttt{(U,U'),R2,U,(R2,R2),F,B',R2,B,F',U,R2} \\ &= \texttt{U,[U',R2],[R2,FB'],U,R2} \end{aligned}$$

This example will limit itself to an evaluation of the edge-permutation, but this is easily extended to the other cases. To analyse this algorithm, we will split it up in 5 parts:

1. `U`
2. `[U',R2]`
3. `[R2,FB']`
4. `U`
5. `R2`

Starting at the first part, which is simply the move `U`, we can check Equations 17 to find out that this corresponds to a permutation

$$U_e^p = (9, 1, 12, 3) \tag{28}$$

When the cube is unscrambled (solved), this means that the set of affected edge-pieces is given by

$$S^e_{\text{aff},1} = \{e_1, e_3, e_9, e_{12}\} \tag{29}$$

Now we will calculate the edge-sets $S^e_X$ and $S^e_Y$, but also $S^e_{X^{-1}}$ and $S^e_{Y^{-1}}$ of the first commutator $[X, Y] =$ `[U',R2]`. We will assume a solved itial edge permutation that was then submitted to `U` (part 1):

$$p_e = (9, 2, 12, 4, 5, 6, 7, 8, 3, 10, 11, 1)$$

$$\begin{aligned} S^e_X &= \{e_1, e_{12}, e_3, e_9\} \\ S^e_Y &= \{e_4, e_7, e_8, e_3\} \\ S^e_{X^{-1}} &= \{e_1, e_{12}, e_4, e_9\} \\ S^e_{Y^{-1}} &= \{e_{12}, e_4, e_7, e_8\} \\ S^e_{\text{aff},2} &= (S^e_X \cap S^e_Y) \cup (S^e_{X^{-1}} \cap S^e_{Y^{-1}}) \\ &= \{e_3\} \cup \{e_{12}, e_4\} \\ &= \{e_3, e_4, e_{12}\} \end{aligned} \tag{30}$$

And indeed, this commutator corresponds to a 3-cycle permutation of the pieces on positions $E_3$, $E_4$ and $E_9$ which at that time, according to $p_e$, are $e_{12}$, $e_4$ and $e_3$ respectively.

$$\texttt{[U',R2]} = (3, 4, 9) \tag{31}$$

For the second commutator $[X, Y] =$ `[R2,FB']` we can astiblish that, assuming the resulting edge-permutation $p_e = (12, 2, 9, 3, 5, 6, 7, 8, 4, 10, 11, 1)$, the union of intersections is the following:

$$\begin{aligned} S^e_{\text{aff},3} &= (S^e_X \cap S^e_Y) \cup (S^e_{X^{-1}} \cap S^e_{Y^{-1}}) \\ &= \{e_7, e_8\} \cup \{e_1, e_4\} \\ &= \{e_1, e_4, e_7, e_8\} \end{aligned} \tag{32}$$

$$\texttt{[R2,FB']} = (7, 8)(9, 12) \tag{33}$$

The next part is again the move `U`, which will now affect the set

$$S^e_{\text{aff},4} = \{e_1, e_4, e_9, e_{12}\} \tag{34}$$

To conclude the analysis we can establish that the move `R2` corresponds to

$$(R^p_e)^2 = (3, 4)(7, 8) \tag{35}$$

Which will at this stage affect the set

$$S^e_{\text{aff},5} = \{e_3, e_7, e_8, e_{12}\} \tag{36}$$

Combining the gathered information, the union of affected edge-sets $S^e_{\text{aff,i}}$ is larger then what we would have hoped for, since the edges $e_1$, $e_4$, $e_7$ and $e_8$ should not be affected by the algorithm as a whole.

$$\bigcup_{i=1}^{5} S^e_{\text{aff,i}} = \{e_1, e_3, e_4, e_7, e_8, e_9, e_{12}\}$$

On closer examination, the permutations cancel out these edge-pieces, leaving only the set $\{e_3, e_9, e_{12}\}$.

$$\begin{aligned} A_{12} &= \texttt{U,[U',R2],[R2,FB'],U,R2} \\ &= (9,1,12,3)(3,4,9)(3,4)(7,8)(9,1,12,3)(7,8)(9,12) \\ &= (9,12,3) \end{aligned} \tag{37}$$

This corresponds to the leftmost image in Figure 13.

## 4.2 Thistlethwaite's 45-Move Algorithm

As mentioned earlier in this Section's introduction, this is a computer algorithm which means that it's almost impossible for a human being to execute it manually. First of all, it requires some processing time to prepare a set of pruning-tables (Section 4.2.2) that are used to solve the cube and cannot be generated manually, since the biggest one has over 1 million entries. Furthermore, even if one would have these tables, it would be very time-consuming to use them to solve the cube manually.

### 4.2.1 General Outlay

The algorithm, which from now on will be referred to by T45, is made up out of 4 stages, in which the following parts are solved:

1. Fix all edge-orientations.
2. Fix the corner-orientations and bring the `LR`-edges in their slice.
3. Bring the corners into their $G_3$-orbit, and move all edges into their slice in an even permutation.
4. Solve the cube by simultaneously fixing both the corner- and edge-permutation.

To assure that each stage stays solved, the number of permitted moves decreases each time we move to a new stage. For each stage, the permitted moves define a group $G_i$. These are given below:

$$\begin{aligned} G_0 &= \langle \texttt{L,R,F,B,U,D} \rangle \\ G_1 &= \langle \texttt{L,R,F,B,U2,D2} \rangle \\ G_2 &= \langle \texttt{L,R,F2,B2,U2,D2} \rangle \\ G_3 &= \langle \texttt{L2,R2,F2,B2,U2,D2} \rangle \\ G_4 &= \langle \texttt{I} \rangle \end{aligned} \tag{38}$$

A cube is said to be in $G_i$ when its state can be aquired (from a solved initial state) by using moves from $G_i$ only. Since any desired move-sequence can be constructed by using moves from $G_0$, any given cube is in $G_0$. This is not true however for its subgroups $G_{i>0}$. The basic idea of T45 is to start with a cube in $G_i$ and then move it to the next group $G_{i+1}$ by only using the moves in $G_i$ and repeat this until it arrives in $G_4$, meaning that it is solved. The groups are constructed in such a way that it is impossible to destroy a previous stage when only using the permitted moves. The number of elements in $G_i$ is called the *order* of $G_i$, $|G_i|$, which decreases as $i$ increases. Taking into account that the double and anticlockwise are counted as one move, thus implicitly included in the groups, the order of each group is given below:

$$|G_0| = 18,\ |G_1| = 14,\ |G_2| = 8,\ |G_3| = 6,\ |G_4| = 0 \tag{39}$$

### 4.2.2 Pruning Tables

Pruning tables are an essential feature in T45, acting as large lookup tables that can tell us what to do in any given situation. They basically hold, for each possible state in $G_i$, the number of moves it takes to get to $G_{i+1}$. This number is called the distance $d$. When $d$ is known for the current state, the algorithm can iteratively apply each move from $G_i$ to the cube and check how the distance changes. If $d$ becomes smaller after performing a certain move, this appearantly was a good idea and we can repeat the search from here. This is then repeated until $d = 0$, which means that we arrived in the next group $G_{i+1}$.

To create a pruning table, one has to start from the solved position and apply each move in $G_i$ to this position. The resulting state is then indexed according to its edge/corner-orientations or edge/corner-permutations (depending on which table is being generated) and these indices $(n, m)$ are used to determine the entry in the table $P_i$ which is now filled with the number '1': $P^i_{n,m} = 1$. Here, the '1' represents the depth $d$ of the current state. When all moves are applied to the initial state, a total number of entries $N \leq |G_i|$ are filled and the process starts all over again. Except this time, all the moves will be applied to all states with $d = 1$ according to $P^i$ and the resulting entries are filled with a '2'.

By repeating this procedure, eventually all possible states will be visited and the entire table is filled with the corresponding distances. This method can thus be used to generate a table from which God's Algorithm can be read by a computer, i.e. the shortest solution to each state. However, for the $3 \times 3 \times 3$ cube, there are simply too many states to generate the table. This is the reason that the algortithm is split up in 4 stages and their corresponding groups $G_i$. The following sections explain the details of each group and how their pruning tables can be generated. A general pseudo-code to generate a pruning-table is listed below, assuming that there is a set of functions that are able to convert each state to a unique index and vice versa.

### 4.2.3 Stage 1: $G_0 \rightarrow G_1$

In this first stage, all edge-orientations are fixed. By definition (Section 2.3), an edge is correctly oriented when it can be solved without making use of (or using an even number of) `U` and `D` turns [3]. Consequently, the edges in the `U` or `D`-face will flip their orientation on each `U` or `D`-turn. This is the reason that in $G_1$, both the `U` and `D`-moves are not allowed anymore (Equation 38). Instead, only double turns of these face are allowed, preserving all edge-orientations.

To move to $G_1$, we need a pruning table which can be easily generated by using an algorithm similar to that listed in Listing 1 (Appendix A). Since we are only considering edge-orientations, we will only need 1 index and $P_1$ will be an $N \times 1$ array instead of an $N \times M$ matrix. To determine the value of $N$, the number of possible edge-orientations is calculated. Without imposing the laws of the cube, this number is $N_{all} = 2^{12} = 4096$ but since the orientation of the final cubie is fixed due to the 2nd law this number reduces to

$$N_1 = 2^{11} = 2048 \tag{40}$$

[3]In this case, 'solved' means back in its original position in the correct orientation, but without taking into account the positions/orientations of all other cubies.

The initial state can be represented by a $1 \times 12$:

$$E = (1, 1, \ldots, 1) \tag{41}$$

which corresponds to an arbitrary state in which all edge-orientations are 'good'. To convert this state to an index $0 \leq n \leq 2047$, we interpret the first 11 entries of $E$ as a binary number and convert it to a decimal number, which in this case would be 2047. When converted back to a binary number, the 12th entry is calculated using the 2nd law.

With these tools at hand, the table can be generated and the distances along with the number of occurrences are listed in Table 2.

| $d$ | $n(d)$ |
|---|---:|
| 0 | 1 |
| 1 | 2 |
| 2 | 25 |
| 3 | 202 |
| 4 | 620 |
| 5 | 900 |
| 6 | 285 |
| 7 | 13 |
| total | 2048 |

Table 2: The number of states in $G_0$ with distance $d$ to $G_1$. It can be seen that the maximum distance is 7, meaning that the edge-orientations of an arbitrary cube can be solved in 7 moves or less. Furthermore, the average number of moves required is $\langle d \rangle = \frac{\sum_i d_i n_i}{\sum_i n_i} = 4.6$.

The pruning-table can now be used to navigate from an arbitrary $G_0$-state to $G_1$ as explained in Section 4.2.2.

### 4.2.4 Stage 2: $G_1 \rightarrow G_2$

All the edge-orientions are now fixed and to keep it this way, the 90° turns of the `U` and `D` face are now prohibited, thus $|G_1| = 14$.

To get to the next stage, we will need to fix all corner-orientations as well and we need to move the `LR`-edges into their slice. This means that the edges which were numbered 9-12 (Section 2.2, Figure 1) have to be placed in positions 9-12. Again, we will need to generate a pruning table that we can use to navigate to such a state. This time, we will need two indices $(n, m)$ to fill the $N \times M$ matrix $P_2$. The $n^{\text{th}}$ row will hold all states corresponding to a *corner-orientation-state* with index $n$, where the $m^{\text{th}}$ row will hold all states corresponing to an *edge-permutation-state* with index $m$.

The initial states will be written as

$$C = (0, 0, \ldots, 0) \tag{42a}$$

$$E = (0, \ldots, 0, 1, 1, 1, 1) \tag{42b}$$

where the 0's in $C$ represent 0-twist of the corners and the 1's in $E$ represent `LR`-slice pieces which are in positions 9-12 in the solved (initial) state.

As a consequence of the 3rd law, the number of possible corner-orientations and edge-permutations (not distinguishing between individual egde-pieces) that can be reached by applying moves in $G_1$ is

$$N_2 \times M_2 = 3^7 \times \binom{12}{4} = 2187 \times 495 = 1082565 \tag{43}$$

To convert all these states to indices, we will again use a binary-to-decimal transformation in the case of the edge-permutation (first 11 elements, which only contain 0's and 1's). This will however, produce an index $0 \leq m' \leq 2047$ which in effect is not practical to use as a direct index since this would make the table larger than it has to be. Instead we will generate a list $(1 \times 495)$ of these indices and the position within this list can function as an column-index to the matrix $P_2$.

The index $m$ can be calculated using a program similar to that listed as the pseudo-code in Listing 2, where also the inverse of this function is given. This can be used to convert an arbitrary corner-orientation to a unique index in the range 0-$v^{n-1}$. These reduce to a binary-to-decimal converter in the case that `v=2`.

The resulting distances are given in Table 3.

| $d$ | $n(d)$ |
|---|---|
| 0 | 1 |
| 1 | 2 |
| 2 | 17 |
| 3 | 134 |
| 4 | 1065 |
| 5 | 8190 |
| 6 | 54694 |
| 7 | 267576 |
| 8 | 560568 |
| 9 | 187204 |
| 10 | 3114 |
| total | 1082565 |

Table 3: The number of states in $G_1$ with distance $d$ to $G_2$.The average number of moves required is $\langle d \rangle = \frac{\sum_i d_i n_i}{\sum_i n_i} = 7.8$.

### 4.2.5 Stage 3: $G_2 \rightarrow G_3$

In $G_2$ we are not allowed anymore to perform quarter turns of both the `F` and `B`-face. This ensures that the corner-orientation remains fixed (Section 2.3) and that the `LR`-slice-edges remain in their slice.

The general outlay from Section 4.2.1 tells us that in order to get into $G_3$, we have to bring the corners into their $G_3$-orbits and place all edge-cubies in their slice, in an even permutation (which means that the corner-permutation is also even). By '$G_3$-orbits' we mean the sets of positions that can be reached by the corner-cubies when we use only moves from $G_3$ (starting from their solved positions). This results in two sets:

$$\text{Orbit}\left(\{\text{c}_1, \text{c}_2, \text{c}_3, \text{c}_4\}\right) = \{\text{C}_1, \text{C}_2, \text{C}_3, \text{C}_4\} \tag{44a}$$

$$\text{Orbit}\left(\{\mathrm{c}_5, \mathrm{c}_6, \mathrm{c}_7, \mathrm{c}_8\}\right) = \{\mathrm{C}_5, \mathrm{C}_6, \mathrm{C}_7, \mathrm{C}_8\} \tag{44b}$$

It should now be clear why Thistlethwaite chose to number the corners in this way.

To generate the pruning table $P_3$, we use the initial states

$$S_C = \{C | C \in G_3\} \tag{45a}$$

$$E = (0, 0, 0, 0, 1, 1, 1, 1) \tag{45b}$$

where $S_C$ denotes the set of corner-permutations in $G_3$ and the 0's and 1's in $E$ denote edge-cubies of the `FB` and `UD`-slice respectively. The `LR`-slice is purposely excluded since these pieces will stay in their slice anyway.

The set $S_C$ must first be generated by starting from a solved corner permutation and then iteratively applying all moves from $G_3$ as one would do when generating a pruning-table. When this is done, 96 different permutations are found, which will now function as the 96 initial states with $d = 0$ in $P_3$. The total size of $P_3$ can be calculated to be

$$N_3 \times M_3 = 8! \times \binom{8}{4} = 40320 \times 70 = 2822400 \tag{46}$$

This corresponds to the 8! possible corner-permutations and the $\binom{8}{4}$ possible edge distributions. To index the different corner-permutations, a code similar to that listed in Listing 3 was used. To index the edge-permutations, we used the same technique as before. Table 4 lists the results of this stage.

| $d$ | $n(d)$ |
|---|---:|
| 0 | 96 |
| 1 | 192 |
| 2 | 864 |
| 3 | 3456 |
| 4 | 11904 |
| 5 | 50880 |
| 6 | 173376 |
| 7 | 358272 |
| 8 | 495168 |
| 9 | 678720 |
| 10 | 692928 |
| 11 | 307392 |
| 12 | 46848 |
| 13 | 2304 |
| total | 2822400 |

Table 4: The number of states in $G_2$ with distance $d$ to $G_3$. The average number of moves required is $\langle d \rangle = \frac{\sum_i d_i n_i}{\sum_i n_i} = 8.8$.

### 4.2.6 Stage 4: $G_3 \rightarrow G_4$

We have now arrived at the final stage, so by restricting ourselves to only double moves, we make sure that the edges stay in their slices and the corners stay in orbit. To finally solve the cube we generate another pruning table, starting from a solved cube:

$$C = (1, 2, \ldots, 8) \tag{47a}$$

$$E = (1, 2, \ldots, 12) \tag{47b}$$

Since the edges are already in their own slices, and the corners are already in orbit, the number of possible states is limited to

$$N_4 \times M_4 = 96 \times \frac{4!^3}{2} = 663552 \tag{48}$$

The results for the final stage are listed in Table 5.

| $d$ | $n(d)$ |
|---|---:|
| 0 | 1 |
| 1 | 6 |
| 2 | 27 |
| 3 | 120 |
| 4 | 519 |
| 5 | 1932 |
| 6 | 6484 |
| 7 | 20310 |
| 8 | 55034 |
| 9 | 113892 |
| 10 | 178495 |
| 11 | 179196 |
| 12 | 89728 |
| 13 | 16176 |
| 14 | 1488 |
| 15 | 144 |
| total | 663552 |

Table 5: The number of states in $G_3$ with distance $d$ to $G_4$. The average number of moves required is $\langle d \rangle = \frac{\sum_i d_i n_i}{\sum_i n_i} = 10.1$.

When all the tables are used to navigate from one phase to the next, then the cube should be solved after finishing this stage. On average, a cube will be solved in around 31 moves using this method. The maximum number of moves required is 45, hence its name: Thistlethwaite 45.

## 4.3 God's Algorithm

We already discussed the impossibility to generate a pruning table that can be used to find God's Algorithm for any state of the $3 \times 3 \times 3$ cube. However, for the $2 \times 2 \times 2$ (Pocket Cube) case this is remarkably easy. This cube has no edge-cubies, so all one needs to do is make a pruning table with 2 indices $(n, m)$ that represent the orientation and permutation of the cubies. To do this, we can use the operators from Section 3.2 because the way the corners are permuted on this cube is the same as on the standard version.

The total size of the table can be calculated to be

$$N \times M = 3^6 \times 7! = 3^7 \times \frac{8!}{24} = 3674160 \tag{49}$$

where the factor 24 comes from the fact that the orientation of the cube is now arbitrary, since there are no centre-pieces to determine the face-color.

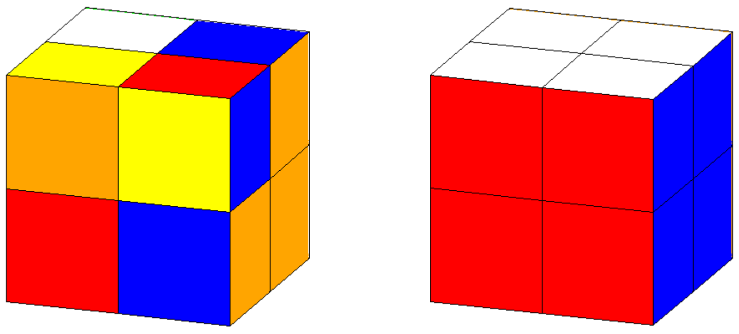
The number of moves is very limited as well, since a `U`-turn has the same effect on the cube as a `D`-turn except for the resulting orientation. We can thus limit ourselves to using only the moves `D,F,R`, thereby keeping the `BL` piece in the same position and orientation.

In order to solve a scrambled cube, we re-orient it so that the `BL`-piece is in the right position and orientation, after which we can navigate through the pruning table that is generated in the usual way (Section 4.2, Appendix A). The results of the search for God's Number are listed below in Table 6.

| $d$ | $n(d)$ |
|---|---:|
| 0 | 1 |
| 1 | 9 |
| 2 | 54 |
| 3 | 321 |
| 4 | 1847 |
| 5 | 9992 |
| 6 | 50136 |
| 7 | 227536 |
| 8 | 870072 |
| 9 | 1887748 |
| 10 | 623800 |
| 11 | 2644 |
| total | 3674160 |

Table 6: The number of states of Rubik's $2 \times 2 \times 2$ Cube that have a distance $d$ to being solved. God's Number appears to be 11, meaning that each cube can be solved in 11 moves or less. There are exactly 2644 states that need this many moves. On average, a cube can be solved in $\langle d \rangle = \frac{\sum_i d_i n_i}{\sum_i n_i} = 8.8$ moves.

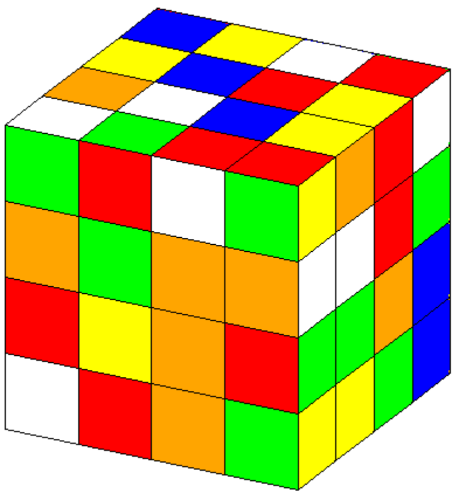
Figure 14: Image of a solved $2 \times 2 \times 2$ cube.



## 4.4 The $4 \times 4 \times 4$ case

The final solving mechanism that we will discuss is one for the $4 \times 4 \times 4$-cube, also known as Rubik's Revenge. The main difference between an ordinary cube

Figure 15: A randomly scrambled $\mathbf{4} \times \mathbf{4} \times \mathbf{4}$ cube: Rubik's Revenge



and the Revenge is the fact that you cannot see which color belongs on which face, since the centre pieces are not fixed. The inner slices can be rotated just as the outer faces and this results in a lot of extra possible positions of the cube: around $7.4 \times 10^45$. This makes it virtually impossible to generate a table from which God's Algorithm can be read, even when great computing power is available. Instead, we choose to move the Revenge to a state that is equivalent to a $3 \times 3 \times 3$ cube by executing the following steps:

1. Restore all centre-pieces (and determine the orientation)
2. Match the edge-pieces
3. Fix the parity of the edge-pairs
4. Apply T45

Before we explain each step in more detail, a new notation has to be introduced to describe the rotations of the inner slices. This notation add lower-case letters to the set of the upper-case letters we used up until now. A lower-case letter denotes a clockwise 90° turn of the slice right behind the face it represents, e.g. `f` denotes a rotation of the slice right behind the `F`-face. Again, the postfixes `'` and `2` are used to specify counter-clockwise or double rotations.

### 4.4.1 Centres

The algorithm that restores the centres looks for a face that has a nearly-solved centre already. The color of these pieces determine the color of this face and the orientation of the cube, i.e. it is determined which colors will go on which faces. The move-sequence that is used to solve each centre is an easy one. Each face is checked for centre-pieces that belong to the top-face and is then brought into a position like the one illustrated in Figure 16: the `bl`-piece is empty and the `dr`-piece needs to move up. Now the algorithm $A_1$ is performed to move the piece to the top-centre:

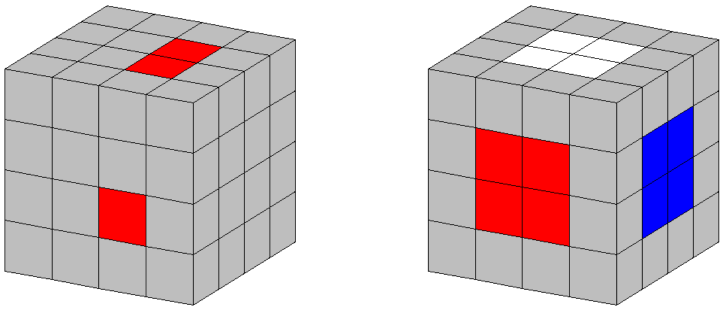
$$A_1 = \texttt{r,U,r}$$

In the case that the piece is on the bottom face in the `rb` position, $A_2$ is performed:

$$A_1 = \texttt{r2,U,r2}$$

This is repeated for each face untill all centres are solved.

Figure 16: To solve the centres, the cube is set-up like the left image after which $A_1$ can be performed to move the `rd` piece to the `U`-face. This is repeated until all the centres are solved.

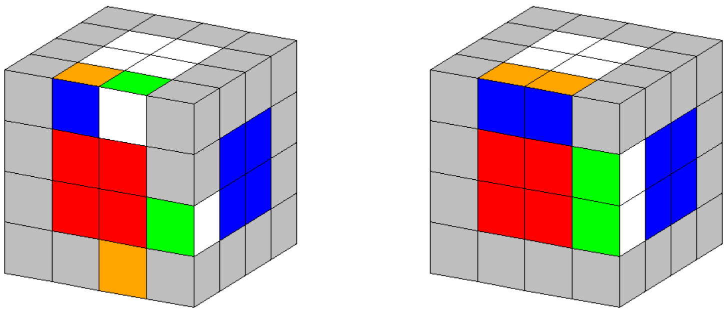


### 4.4.2 Edge-Pairs

To reach a state that is equivalent to that of an ordinary cube, the edges have to be paired. We need to place two pairs of matching edges in the `FUl`/`FDr` and `FUr`/`FRd` positions and then apply $A_3$ to join both pairs of edges as shown in Figure 17.

$$A_3 = \texttt{r,U',R,U,r',U',R',U}$$

Figure 17: The edges are paired two at the time. The left image shows how the edges are to be set-up. Be careful to use face-turns only in order to keep the centre pieces on their solved positions.



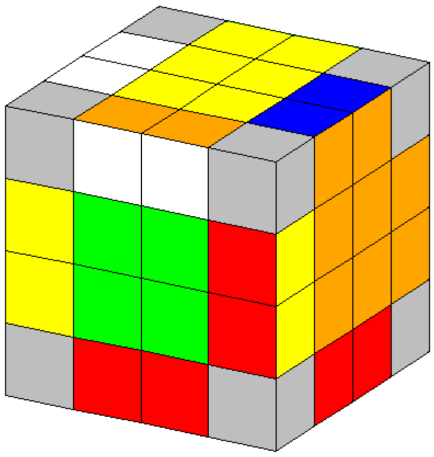
It could happen that there are only 2 unmatched pairs left on the cube, while one needs at least 3 unmatched pairs to perform the algorithm as mentioned above. In that case, $A_4$ can be applied to make sure that there are 3 unmatched pairs again to work with.

$$A_4 = \texttt{U2,r,U2,r,U2,r,U2,r,U2,r,U2}$$

Another possibility is that the pairs of edges are already adjecent to one another, in which case it is impossible to move the pieces into a position like that in Figure 17 without using slice-moves (since these are necessary to separate the edge pieces). In this case, applying $A_3$ will only match one pair instead of two.

Repeating this procedure will eventually pair-up all the edges. An example of a resulting state is shown in Figure 18.

Figure 18: Possible result after all the edges have been paired.



### 4.4.3 Fixing the Parity

On first sight, the cube now looks like a regular $3 \times 3 \times 3$ cube but there are situations in which it cannot yet be solved using one of the familiar methods. As described in Section 2, there are certain laws that constrain the number of possible states. The $1^{\text{st}}$ and $2^{\text{nd}}$ law tell us that only half the edge orientations and permutations can be reached by using genuine moves (that is, without taking the cube apart). However, when converting a $4 \times 4 \times 4$ cube to a $3 \times 3 \times 3$ analogue, violations of these laws may occur. There are two possible such violations:

1. Two edges are swapped, i.e. the edge parity is odd (violation of the $1^{\text{st}}$ law).
2. One of the edges is flipped, i.e. $\sum\limits_i o_{c,i} \mod 2 = 1$ (violation of the $2^{\text{nd}}$ law).

Before we can apply T45 (or any other method), these exeptions have to be fixed by using the following algorithms (in the same order as above):

$$A_5 = \texttt{d2,r2,D2,d2,r2,D2,r2}$$

$$A_6 = \texttt{r2,R2,B2,L',D2,l',D2,r,D2,r',D2,F2,r',F2,l,L,B2,R2,r2}$$

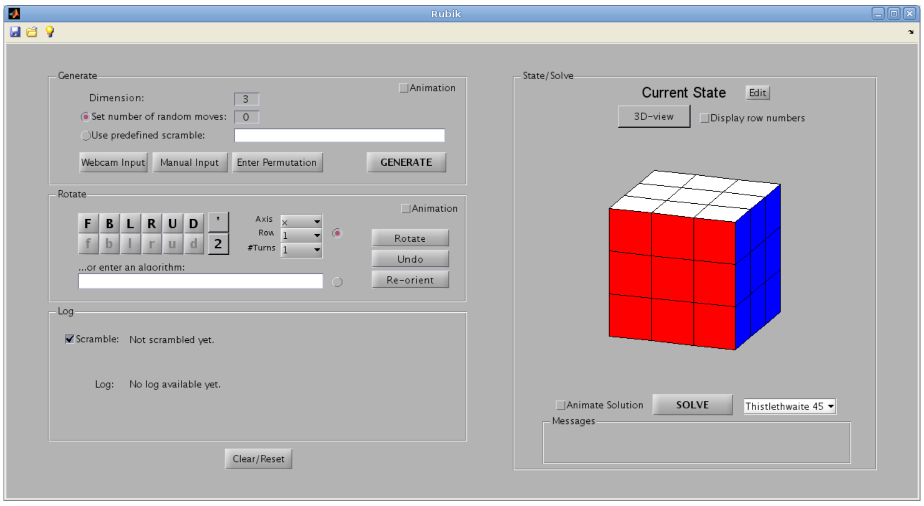
### 4.4.4 Solving the Cube

The current state can now be solved using any method to solve a $3 \times 3 \times 3$ cube. In the Matlab implementation, the most efficient method was chosen which is T45. This method was therefore named 423T45 (4 to 3, T45).

## 5 A Practical Manual to the Matlab Implementation

What will now follow, is a manual to the Matlab implementation, omitting the theory behind it, since that can be derived from previous sections. We will start by introducing the general interface of the program, as is shown in Figure 19[4]. The window contains 4 panels that each contain a number of controls that

Figure 19: General interface of the ‘Digital Rubik’s Cube’-program.



Dimension: 3
[✓ Set number of random moves:] 0
Use predefined scramble:
Axis x
Row 1
#Turns 1
...or enter an algorithm:
[✓ Scramble:] Not scrambled yet.
Log: No log available yet.
Thistlethwaite 45

can be used to generate, manipulate or solve the cube which is displayed in the rightmost panel.

### 5.1 Generating a Cube

For starters, we need to generate a cube. By default, a standard $3 \times 3 \times 3$ cube is active and on display which can immediately be used. However, the ‘Generate’ panel allows the user to either generate a random cube, or generate a cube according to preset conditions.

1. To generate a random cube, one has to enter the dimension $d$ and the number of random moves $N$ that will be applied to scramble the cube. Both must be an integer value greater than 0. When these parameters are set, the program does the following:
   (a) Generate a solved cube of dimension $d$ (Section 3.1).
   (b) Generate a list of $N$ random moves.
   (c) Apply an optimalisation algorithm to this sequence to combine subsequent moves if possible $\rightarrow n$ remaining moves.
   (d) If $n < N$, add $N - n$ moves to the cube and go to (c).

[4]This is the interface as seen on a Linux-machine and may deviate slightly from the image on a Windows machine.

(e) Apply the moves to the cube.

2. The second possibility is to generate a cube using a predefined scrambling sequence. This is done by entering the desired scrambling sequence in the textbox. Note that still the dimension has to be set in order to tell the program to what kind of cube it has to apply the moves. This brings us to the notation conventions that are used when specifying moves to the program. For cubes with $d \leq 4$, the standard notation can be used, i.e. `F,f,B,b,L,l,R,r,U,u,D,d` with the prefixes `'` and `2` to indicate counter-clockwise or double moves.

   When working with higher-dimension cubes, the notation as described in Section 3.1 is used. In this notation, a move is represented by the axis, the row-number and the number of moves e.g. `x32` for a double rotation of the 3$^{\text{rd}}$ row around the $x$-axis. This notation might be confusing because it is unlear which face on the cube is represented by which axis/row combination. To figure this out, I refer the reader to Figure 3 and the 'Display row numbers' checkbox in the right panel.

   Both conventions may be used on any cube, but will always be converted by the program to the appropriate notation according to the cube-dimension.

3. When a real-world cube (with unknown scramble) needs to be entered in the program, this can be done in two ways: by webcam or manually.

   The webcam-feature naturally requires a webcam being connected and on top of that it needs Matlab's Image Processing Toolbox to be installed. When this is the case, clicking the 'Webcam Input'-button should initiate a live webcam image in the right panel. Instructions below will indicate which face is to be shown before clicking the 'Capture'-button. Clicking this button will cause the program to try to find the cube in the image and determine its facelet-colors. When this is done succesfully for all faces, the digital version of the cube is stored and can be solved or manipulated digitally.

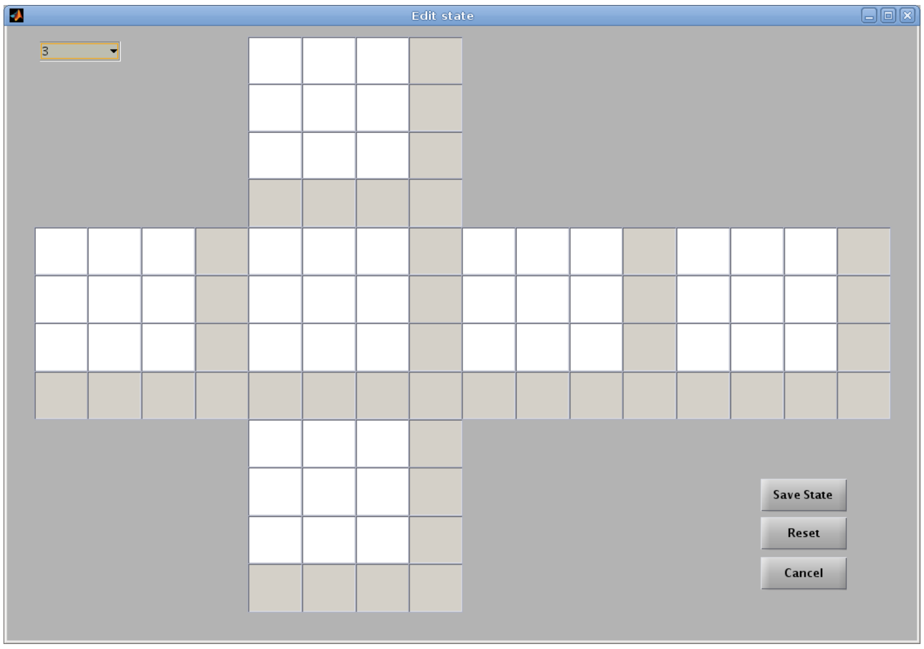
4. The 'Manual Input'-button opens a new window, showing a 2D-map of the cube: Figure 20. This can be used to input cubes of dimension $d = 2, 3, 4$. The user has to specify the colors of each face by activating the corresponding facelet and entering the color by its first letter: `R,B,O,G,Y,W`. When the 'Save State' button is clicked, the validity of the cube is checked and if valid, the cube is succesfully entered in the program. If not valid, the user may choose to continue anyway or try to find the error until the cube is valid.

5. The last way to enter a state is by specifying the permutations and orientations of the cubies. This feature is only available for the standard $3 \times 3 \times 3$ cube and can also be used to read/edit the permutation/orientations of the current cube.

## 5.2 Manipulating the Cube

The 'Rotate'-panel contains all the controls that are needed to manipulate the current cube. The easiest way to do so is by clicking the buttons that hold the

Figure 20: The Manual Input window for inputting a $3 \times 3 \times 3$ cube. Also, cubes of dimension 2 or 4 can be entered using this method.



standard moves. This will work on all cubes, but ofcourse cannot be used to rotate the innermost slices of cubes with $d > 4$. When rotations of this kind are desired, one has to specify again the axis, row number and number of rotations using the drop-down menus.

It is also possible to execute an entire move sequence at once, as is often the case when solving a cube. This is done by entering the sequence (delimited by `,`) in the text-box and clicking 'Rotate'. The same notation conventions are used as mentioned above in Section 5.1.

The 'Animation'-checkbox toggles the option to animate each rotation, but this may only be checked when $d \leq 5$, since higher dimensions will mostly produce slow and annoying animations that do not contribute to the clarity anyway.

The 'Re-orient' button will cause the camera-position (which can be altered by clicking the '3D-view' button) to reset to its default value and, if possible, re-orient the cube to its default orientation: Red = Front, White = Up. For scrambled cubes of even dimension, this is not possible since there is no fixed centre-piece to determine the face-colors.

## 5.3 Solving the Cube

The right panel shows the current cube-state and holds the most important button of all: 'Solve'. When this button is clicked, the cube is solved using the selected solving-method. This can be one of five methods:

1. God's Algorithm (for $d = 2$)

2. Thistlethwaite 45 (for $d = 3$)
3. Layer-by-Layer (for $d = 3$)
4. 423T45 (for $d = 4$)
5. Inverse Scramble (for known scramble, all dimensions)

After the cube has been solved, the user is prompted the option to see the solution algortithm which will be placed in a temporary text-file which can then be saved if desired.

The 'Edit' button at the top of this panel allows the user to edit the current state at any point. However, one has to be careful not to enter an unsolvable state which will be recognised immediately for $d = 2, 3$ but not for $d = 4$. When trying to solve an insolvable state, the program may produce errors or enter an infinite loop. Bottom line: be careful when using the 'Edit'-button

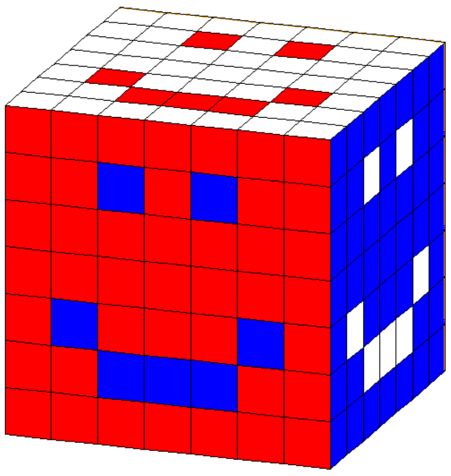
## References

[1] http://www.jaapsch.net/puzzles/

[2] http://www.ryanheise.com/cube/

[3] http://kociemba.org/cube.htm

[4] http://cube20.org/



# Appendices

## A Listings

Listing 1: Pseudo code for generating a prune-table

```
P = empty NxM-table
moves = array of available moves
C = initial corner-state
E = initial edge-state
n = state2index(C)
m = state2index(E)
d = 0
P(n,m) = d
while P not completely filled
  d = d+1
  [x,y] = indices of entries in P with value d-1
  for i from 1 to length(x)
    currentC = index2state(x)
    currentE = index2state(y)
    for j from 1 to length(moves)
      newC = apply move j to currentC
      newE = apply move j to currentE
      n = state2index(C)
      m = state2index(E)
      if P(n,m) is empty
        P(n,m) = d
      end if
    end for
  end for
end while
```

Listing 2: Conversion Functions

```
function index = orientation2index(or)

  n = length(or)
  v = 3
  index = 0
  for i from 1 to n-1
      index = index * v
      index = index + or[i]
  end

function or = index2orientation(index)

  s = 0;
  or = 1xn array
  for i from n-1 to 1
      or[i] = index mod v
      s = s - or[i]
      if s < 0
      s = s + v
      end
      index = (index-or[i])/v
  end
  or[n] = s
```

Listing 3: Conversion Functions

```
function index = permutation2index(perm)

  n = length(perm)
  index = 0
  for i from 1 to n-1
      index = index * (n + 1 - i)
      for j from i+1 to n
      if perm[i] > perm[j]
          index = index + 1
      end
      end
  end

function perm = index2permutation(index)

    perm = 1xn array
    perm[n] = 1;
    for i from n-1 to 1
        perm[i] = 1 + (index mod (n-i+1))
        index = (index - (index mod (n-i+1)))/(n-i+1);
        for j from i+1 to n
            if perm[j] >= perm[i]
                perm[j] = perm[j]+1;
            end
        end
    end
```