An Introduction to the Cubing Package

Alec Stephenson
Melbourne, Australia
Version 1.0-3, 2018-03-25

December 6, 2017

Summary

This document gives an brief non-technical introduction to the R package **cubing**. The package contains functions for analyzing, visualizing and solving the Rubik's cube. The solvers are based on the methodology of Herbert Kociemba. For speed reasons the solvers are written in C, largely based on code developed by Maxim Tsoy. The 3D plots and animations use an interface to OpenGL.

The definitive help for any function can be seen by typing a question mark followed by the function name, such as `?move` for the `move` function. An exception is for the operator `%v%`, which must also be surrounded by quotation marks, so use `?"%v%"` to read the help on that. The help file will be more detailed than the information given in this document. Using `examples(move)` will run the code in the examples section of the help file for the `move` function.

I will always use double quote marks rather than single quote marks when using character strings, because a single quote mark is used for prime moves such as `U'` and `F'`. This means that if you use single quote marks, you will need to escape the prime with a backslash. Double quote marks are therefore simpler to use. It is assumed that you know standard move notation for cubing. The package also accepts variants for input, for example `U U1 U' U1' U3 U3' U2` and `U2'` are all valid moves.

The move `U3` represents three quarter-turns in the clockwise direction, which gives the same cube as the anti-clockwise quarter turn `U'` (or `U1'`), but the animations will display the different turn directions. This similarly applies to the `U3'` and `U` (or `U1`) moves, and to the half-turn moves `U2'` and `U2`.

In this package we define middle slice moves by `E = D'Uy'`, `M = L'Rx'` and `S = B'Fz'`. The `S` move is therefore in the opposite direction to the more commonly used definition. It is re-defined here for consistency with the direction of the `x y z` rotations. Wide moves can be denoted by either lower case letters or by `w` notation. For example, `u` and `Uw` are equivalent: both are equal to `UE'= Dy`.

If you use this package, then it would be good to also reference it. To cite this package or this manual please type `citation("cubing")` and use the resulting citation.

Contents

1 Getting Started: Creating and Viewing

1.1 Creating a Pretty Pattern

To get started we will create a cube. There are two types of cube representations in the package: stickerCubes and cubieCubes. For now, we just use cubieCubes. The function `getCubieCube` creates a cubieCube with a pretty pattern. You can call the function with no arguments, as in the code below. And don't forget, the **cubing** package also needs to be loaded with the function `library` for you to use it. If `library` does not work, then it probably isn't installed: typing `install.packages("cubing")` and hitting the return key should do the job.

```
> library(cubing)
> hello.cube <- getCubieCube()
> hello.cube
```

The code creates a cubieCube object called `hello.cube` and prints it out to the screen. If `getCubieCube` is used with no arguments, the object is simply the solved cube, which is a pretty pattern, but is not particularly interesting. So instead, we can create some other[1] patterns. I'll just use three here, but there are approximately seventy different patterns that you can choose from.

```
> aCube <- getCubieCube("Superflip")
> bCube <- getCubieCube("EasyCheckerboard")
> cCube <- getCubieCube("HenrysSnake")
```

[1]The patterns and their names are taken from the ruwix.com website.

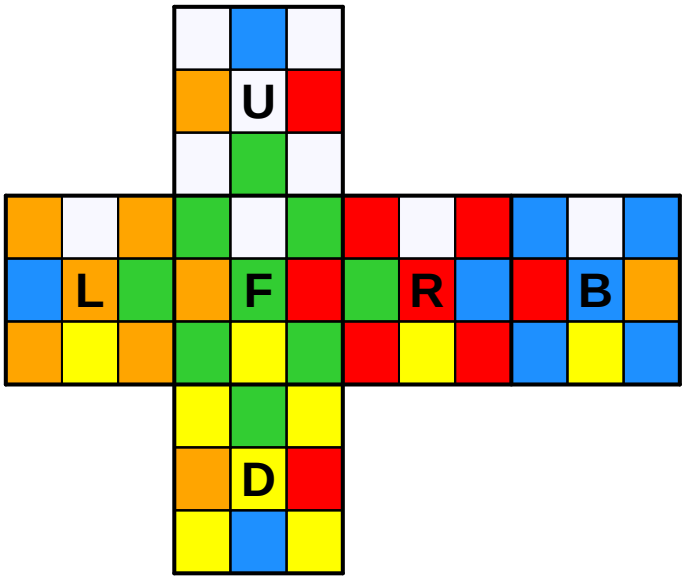

Figure 1: A 2D plot of the superflip, produced using `plot(aCube)`. For a 3D interactive plot, use `plot3D(aCube)`.

There isn't much point creating pretty patterns that you can't see. There are two ways of displaying a cube. The first is the `plot` function[2] which creates a flat 2D plot. The second is to use the `plot3D` function with creates an interactive 3D plot that you can use your mouse to rotate and zoom. For example, the superflip cube in Figure 1 can be displayed using `plot(aCube)`. For the 3D interactive plot, use `plot3D(aCube)`.

1.2 Sticker Colors

The R option `cubing.colors` controls the sticker colors used by default in the `plot` and `plot3D` displays. The default colors are given by

```
> getOption("cubing.colors")

[1] "ghostwhite" "red"        "limegreen"  "yellow"     "orange"
[6] "dodgerblue"
```

[2]The `plot` and `plot3D` functions are called generic functions. This means that if you want to read the help files you will need to use `?plot.cube` and `?plot3D.cube`.

The ordering of the faces corresponding to the six colors is U R F D L B, so the first color is the color of the centre sticker on the U face, and the second color is the color of the centre sticker on the R face. Initially (i.e. prior to any whole cube rotation) the U face is ghost white and the F face is lime green, corresponding the the WCA (World Cubing Association) scrambling orientation.

You can change the initial scrambling orientation by specifying a different order for the color vector; for example yellow on U and blue on F.

```
> mycol <- c("yellow", "dodgerblue", "red",
+            "ghostwhite", "limegreen", "orange")
> options(cubing.colors = mycol)
```

Or you can use a Japanese color scheme. Colors on opposite faces are always three apart within the ordering U R F D L B.

```
> jcol <- c("ghostwhite", "red", "limegreen",
+           "dodgerblue", "orange", "yellow")
> options(cubing.colors = jcol)
```

Or maybe just use some different colors[3] entirely. You can specify[4] any R color name: type `colors()` to see all the options.

```
> acol <- c("black", "darkred", "darkgreen",
+           "yellow", "orange", "purple")
> options(cubing.colors = acol)
```

However we will only use the default colors for this document:

```
> ocol <- c("ghostwhite", "red", "limegreen",
+           "yellow", "orange", "dodgerblue")
> options(cubing.colors = ocol)
```

1.3 Creating a Random State Cube

We have seen how to create a cube with a pretty pattern, so here we will do the opposite and create a cube where the state of the cube is entirely[5] random. This is done with the function `randCube`. If called with no arguments, it produces a single cube, but if the first argument is the integer n it produces a list of n cubes.

```
> aCube <- randCube()
> plot(aCube)
> plot3D(aCube)
```

[3]This approximately corresponds to the Autumn color scheme from www.speedcube.com.au, where you can purchase lots of cubes and stickers of different colors.

[4]For the color expert you can also specify any rgb hex code: for example `"#FF0000"` is exactly the same as `"red"`. There are currently 502 distinct named colors in R, but there are 16777216 distinct rgb colors. The six sticker colors are not required to be distinct, because they are only used for display purposes.

[5]More precisely, every solvable cube state is equally likely to occur. There are 43252003274489856000 solvable cube states and so every state has a 1 in 43252003274489856000 chance of occurring.

1.4 Creating a Moves Cube

Another way to create a cube is to construct one that represents a move sequence. For example, consider the move sequence R U' F2. Move sequences can be specified using a single character string which may contain any amount of white space, for example `"RU'F2"` and `"R U' F2"` are both okay. You can also specify a character vector of moves rather than a single character string, which may be more convenient for reading in moves from a text file or the web.

The function `getMovesCube` creates a cube corresponding to a move sequence. The code below creates the cube `aCube` corresponding to R U' F2.

```
> aCube <- getMovesCube("RU'F2")
```

The cube `aCube` is the result of applying R U' F2 to a solved cube. If the move sequence `mv` represents a scramble, then `getMovesCube(mv)` is the scrambled cube.

More generally, cubes produced using `getMovesCube(mv)` can represent the application of the move sequence `mv` to any cube. The following code demonstrates this.

```
> rCube <- randCube()
> rCube <- rCube %v% aCube
```

Here we generate a random cube `rCube` and apply the move sequence R U' F2 using the cube composition[6] operator `%v%`.

So `getMovesCube` produces a cube representation of a move sequence that can then be applied to any cube. A disadvantage of this approach is that the move sequence cannot include whole cube rotations, because rotations do not work this way. If you want to apply a move sequence that includes rotations, you will need to use the `move` function: see Section 3.1.

1.5 Creating Even More Pretty Patterns

Another use of the cube composition operator `%v%` is to create even more pretty patterns by combining the existing pretty patterns.

```
> aCube <- getCubieCube("Anaconda") %v% getCubieCube("FourSpots")
> bCube <- getCubieCube("FourSpots") %v% getCubieCube("Anaconda")
> plot3D(aCube)
> plot3D(bCube)
```

[6]For those that know about matrices, the cube composition operator `%v%` works in a similar way to the matrix multiplication operator `%*%`. Both operators are associative but not commutative. The solved state corresponds to the identity matrix, and each cube has a unique inverse given by the `invCube` function.

1.6 Creating a Random Moves Cube

The function `randMoves` is very similar to `randCube` but it creates random move sequences[7] rather than random cubes. The second argument to `randMoves` gives the length of each move sequence, which by default is 20. By combining this with `getMovesCube` we can create a different type of random cube which is constructed from random moves.

```
> rCube <- getMovesCube(randMoves(1, nm = 22))
```

The above code creates a random cube using a random sequence of 22 moves. Unlike the random state cubes generated with `randCube`, each state will not be equally likely[8] to occur.

1.7 Creating Any Cube

Any software about the Rubik's cube needs to provide a way of reading in the actual state of a real cube. In this package you can use the `cubieCube` function. But first, run the following code which produces Figure 2.

```
> plot(getCubieCube(), numbers = TRUE, blank = TRUE)
```

The `cubieCube` function takes a character string (or a character vector) argument. The character string can contain any amount of white space and can only contain the letters U R F D L B which correspond to the colors of the centre stickers on the respective faces. The sticker ordering is given by the scheme in Figure 2.

```
> aCube <- getCubieCube("Wire")
> bCube <- cubieCube("UUUUUUUUU RLLRRRLLR BBFFFFFBB
+                    DDDDDDDDD LRRLLLRRL FFBBBBBFF")
```

The two cubes `aCube` and `bCube` produced by the code above are identical. The blocks of nine letters in the call to `cubieCube` respectively represent the sticker colors on the U R F D L B faces.

The fifth letter of every block of nine corresponds the the centre sticker and so every fifth letter is respectively fixed at U R F D L B. Because the centre stickers are fixed, you can just omit them all and use eight letters for each face. For example, the cube `cCube` generated below is exactly the same as `aCube` and `bCube`.

```
> cCube <- cubieCube("UUUUUUUU RLLRRLLR BBFFFFBB
+                    DDDDDDDD LRRLLRRL FFBBBBFF")
```

[7] For more on move sequences, see the functions `invMoves`, `rotMoves` and `mirMoves`, which invert, rotate and mirror move sequences. See also `moveOrder`, which calculates the order of a move sequence.

[8] There will be a non-zero chance of occurrence for every state provided that the number of moves is 20 or more.

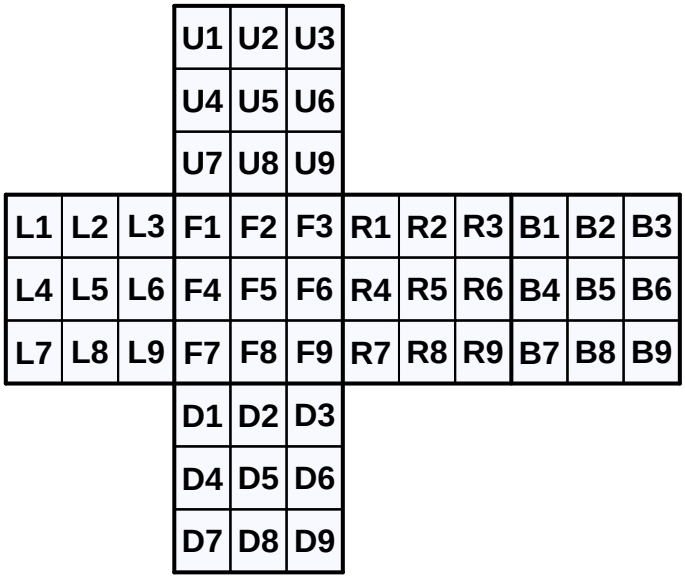

Figure 2: A 2D plot of the sticker labelling scheme.

A large amount of error checking is performed to ensure that the created cube is stickered correctly. Below are some incorrectly specified character strings and the errors that they respectively produce. Although the cube needs to be stickered correctly, it does not need to be solvable: see Section 2.2.

```
> cubieCube("UUUUUUUUU RLLRRRLLR BBFFFFFBB DDDDDDDDD LRRLLLRRL FFBBBBBF")
> cubieCube("UUUUUUUUU RLLRRRLLR BBFFFFFBB DDDDDDDDD LRRLLLRRL FFBBBBBFU")
> cubieCube("UUUUUUUUU RLLRRRLLR BBFFFFFBB DDDDDDDDD LRRLBLRRL FFBBLBBFF")
> cubieCube("UUUUUUUUU RLLRRRLLR BBFFFFFBB DDDDDDDDD LRRLLLRRF FFBBBBBFL")
> cubieCube("UUUUUUUUU RLLRRRLLR BBFFFFFBB DDDDDDDDD LRRLLBRRL FFBBBLBFF")
```

```
Error in stickerCube(string) : string of incorrect length
Error in stickerCube(string) : must have nine of each color URFLBD
Error in stickerCube(string) : centre pieces are incorrect color
Error in stickerCube(string) : corner piece has invalid stickers
Error in stickerCube(string) : edge piece has invalid stickers
```

2 Cube Representations and Solvability

2.1 Cube Representations

There are two data objects in the package, called stickerCubes and cubieCubes, that both represent cubes. Some package functions, such as the plotting functions, will work with either representation. The cubieCube representation is better for mathematical operations, so some functions will return an error[9] for stickerCubes.

The functions `randCube` and `getMovesCube` will return a cubieCube by default, but will return a stickerCube upon setting the argument `cubie` to `FALSE`. The functions `getCubieCube` and `cubieCube` have equivalent functions `getStickerCube` and `stickerCube` for creating stickerCubes.

The code and output below shows a random cube under each representation. The conversion functions `as.stickerCube` and `as.cubieCube` can be used to convert a cube from one representation to the other. The `is.stickerCube` and `is.cubieCube` functions perform tests of cube type.

```
> aCube <- randCube()
> aCube
```

```
$cp
URF UFL ULB UBR DFR DLF DBL DRB
  3   5   7   1   4   6   2   8

$ep
```

[9]For speed reasons there is no explicit error checking for `%v%`, so although it will return an error, the error message will be somewhat obscure. Just remember not to use `%v%` with stickerCubes.

```
FR FL BL BR UR UF UL UB DR DF DL DB
 1  3  7 12  4  6  2  8 10  5  9 11

$co
URF UFL ULB UBR DFR DLF DBL DRB
  0   1   0   2   0   1   2   0

$eo
FR FL BL BR UR UF UL UB DR DF DL DB
 0  0  1  0  0  1  1  0  0  0  1  0

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

```
> as.stickerCube(aCube)

U1 U2 U3 U4 U5 U6 U7 U8 U9 R1 R2 R3 R4 R5 R6 R7 R8 R9 F1 F2 F3 F4 F5
 D  U  R  L  U  B  R  F  U  L  R  U  R  R  B  R  F  R  D  U  B  B  F
F6 F7 F8 F9 D1 D2 D3 D4 D5 D6 D7 D8 D9 L1 L2 L3 L4 L5 L6 L7 L8 L9 B1
 F  L  R  B  F  U  U  R  D  D  F  D  D  B  F  F  U  L  L  U  D  D  F
B2 B3 B4 B5 B6 B7 B8 B9
 B  L  D  B  L  B  L  L
```

A stickerCube object is just a named vector, where the names correspond to the sticker location given in Figure 2. The values in the vector are character strings (printed without quotes) representing the colors U R F D L B.

A cubieCube object is a list giving vector components for the corner permutation (cp), corner orientation (co), edge permutation (ep) and edge orientation (eo). The names within each vector represent the locations of corners and edges. For orientation, a zero represents a correctly oriented[10] cubie, a one represents a flipped edge or a clockwise twisted corner, and a 2 represents an anti-clockwise twisted corner.

A cubieCube object also contains a spor (spatial orientation) vector. This tracks the location of the fixed centre pieces. It is changed only when a whole cube rotation is performed, or when a middle slice (E, M or S) move is performed. The purpose of the spor component is to ensure that the colors used for plotting are consistent before and after rotations. If the component did not exist, then the cube plots would need to be recolored[11] after rotations.

2.2 Solvability

An important point about our cube objects is that they are designed to hold both solvable and unsolvable cubes. If you just take the cubies of your cube apart and reconstruct it,

[10] We use the most common definition for orientation, so a correctly oriented edge can be solved in <U R F2 D L B2> and a correctly oriented corner can be solved in <U R2 F2 D L2 B2>.

[11] The stickerCube representation does not contain spatial orientation information because the U5 value must be U, the R5 value must be R, and so on.

then the cube may not be solvable, but you can still represent it with a stickerCube or a cubieCube. It is actually very useful to be able to represent unsolvable cubes. Even the solver can be usefully applied to an unsolvable cube, if the target state is also unsolvable (see Section 6).

The functions `is.solvable` and `is.solved` test for solvable and solved cubes, and by default simply return `TRUE` or `FALSE`, but if the argument `split` is `TRUE` they give a little more information.

There is quite a lot happening in the code line below, but the output is just a whole number. If the argument `solvable` is set to `FALSE`, then `randCube` will produce a cube (or list of cubes) that is randomly constructed from the cubies but is not necessarily solvable. The R function `sapply` applies another function, in this case `is.solvable`, to every element of a list, and the `sum` function will add up the number of `TRUE` results.

The code line therefore simulates 100 random cube constructions, and works out how many of those 100 are solvable. This number is random, but the chance of any simulated cube being solvable is 1 in 12, so we expect[12] it to be close to $100/12 = 8.3$.

```
> sum(sapply(randCube(100, solvable = FALSE), is.solvable))

[1] 7
```

The `is.solvable` function also has a `split` argument; if this is `TRUE`, then it returns a logical value for each of three components: permutation parity[13], edge orientation and corner orientation. For a cube to be solvable, all three must be `TRUE`. For the code below, there is a 1 in 2 chance that permutation parity is `TRUE`, there is also a 1 in 2 chance that edge orientation is `TRUE`, and there is a 1 in 3 chance that corner orientation is `TRUE`.

```
> rCube <- randCube(1, solvable = FALSE)
> is.solvable(rCube, split = TRUE)

parity     co     eo
  TRUE  FALSE   TRUE
```

2.3 Functions Impacting Solvability

There are some functions in the package that can create unsolvable cubes from solvable cubes, and vice-versa. The `flipEdges` and `twistCorners` functions allow you to flip any number of edges or twist any number of corners in any direction. The `cycleEdges` and `cycleCorners` functions perform permutation cycles on edges or corners. If a k-cycle is performed where k is even[14], then a solvable cube will become unsolvable. There also

[12] Formally, the result is a binomial random variable with parameters $n = 100$ and $p = 1/12$, so the expectation is $np = 8.3$.

[13] You can use the function `parity` to calculate the sign (odd or even) of the corner and edge permutations. Type `?parity` for more details. For the cube to be solvable the corner and edge permutation signs must be the same (both odd or both even).

[14] This is because a k-cycle can be written as $k - 1$ transpositions (swaps, or 2-cycles), and each transposition changes the permutation sign.

exists operators %e% and %c% which are similar to %v% but act only on the edges and corners respectively. Both A %e% B and A %c% B can be unsolvable, even if A and B are solvable.

The invCube function, which calculates the inverse cube state, does not impact solvability, because the inverse of a solvable cube is always solvable, and the inverse of an unsolvable cube is always unsolvable.

It can be more complicated to determine when an unsolvable cube becomes solvable. For example, the composition %v% of two unsolvable cubes can be solvable or unsolvable. This is easy to see: if A is unsolvable then the inverse invCube(A) is also unsolvable, but A %v% invCube(A) and invCube(A) %v% A are by definition both equal to the solved state which is obviously solvable.

3 Move a Cube and Generate Scrambles

3.1 Move a Cube

In Section 1.6 we saw one way to generate moves using the %v% operator with getMovesCube, but the move function is more general because it allows for rotations (x, y and z), middle slice (E, M and S), and wide (Uw, Rw, Fw, Dw, Lw and Bw) moves. The first argument of the move function is the cube to be moved, and the second is the move sequence.

For our example we will use the 4.59 second world record of SeungBeom Cho. We need first to read in the scramble and a reconstruction of the solve. To do this we use the R function scan and just copy and paste from http://www.cubesolv.es/. You could also save them as text files and specify the filenames as an argument to scan. We immediately construct the scrambled cube by placing the first call to scan (which reads the scramble sequence) within the getMovesCube function.

```
> aCube <- getMovesCube(scan(what = character()))
1: D2 F2 U F2 D R2 D B L' B R U L R U L2 F L' U'
20:
Read 19 items

> mv <- scan(what = character(), comment.char = "/")
1: x2 // inspection
2: D' R' L2' U' F U' F' D' U' U' R' // XXcross
13: y' R' U' R // 3rd pair
17: y' R U' R' U' R U R' // 4th pair
25: U' R' U' F' U F R // OLL(CP)
32: U' // AUF
33:
Read 32 items
```

Alternatively, the function read.cubesolve can be used to read from the cube solves website directly. The id for the SeungBeom Cho world record is 4995, so we pass this id to the function. The returned value is a list with the scramble, solution and description.

```
> cho <- read.cubesolve(4995)
> cho
$scramble
 [1] "D2" "F2" "U"  "F2" "D"  "R2" "D"  "B"  "L'" "B"  "R"  "U"  "L"  "R"
[15] "U"  "L2" "F"  "L'" "U'"

$solution
 [1] "x2"  "D'"  "R'"  "L2'" "U'"  "F"   "U'"  "F'"  "D'"  "U'"  "U'"
[12] "R'"  "y'"  "R'"  "U'"  "R"   "y'"  "R"   "U'"  "R'"  "U'"  "R"
[23] "U"   "R'"  "U'"  "R'"  "U'"  "F'"  "U"   "F"   "R"   "U'"

$description
[1] "4.59 WR 3x3 solve by  SeungBeom Cho at ChicaGhosts 2017"

> aCube <- getMovesCube(cho$scramble)
> mv <- cho$solution
```

Now we have the scrambled cube `aCube` and the move reconstruction `mv`, we can use the `move` function to apply the moves to the cube.

```
> result <- move(aCube, mv)
> is.solved(result)

[1] TRUE
```

This confirms that the resulting cube is solved. We will continue this example in Section 4 when we talk about animations and flick books.

3.2 Generate Scrambles

The function `scramble` generates a scrambling sequence. The function will by default generate a random moves scramble, but will generate a random state scramble if the argument `state` is `TRUE`. For a random moves scramble the number of moves can be specified using the argument `nm`.

A random state scramble operates by generating a random cube with the `randCube` function, solving the random cube, and returning the inverse[15] of the resulting move sequence. For a random state scramble there are a number of arguments that are passed on to the solver, including the `maxMoves` requirement which specifies the maximum number of moves allowed.

Three random state scrambles are generated below. By default, the maximum number of moves allowed is 24. Using this upper limit, you can generate 100 random state scrambles in a few seconds.

```
> scramble(3, state = TRUE)
```

[15] The `solver` function has an `inv` argument that can be set to `TRUE` to return the inverse move sequence.

```
[[1]]
 [1] "B2" "L2" "F2" "L2" "D'" "B2" "U2" "B2" "L2" "D'" "R2" "D"  "R2"
[14] "U"  "F"  "L'" "B2" "D"  "L2" "D'" "F"  "L"  "B'" "U'"

[[2]]
 [1] "U2" "F2" "D'" "L2" "F2" "U'" "R2" "D2" "R2" "U"  "B2" "U"  "L2"
[14] "R"  "B'" "U'" "R"  "B2" "R'" "B"  "D'" "B'" "F'" "U'"

[[3]]
 [1] "U'" "B2" "U'" "F2" "D"  "R2" "U"  "F2" "D2" "B2" "L2" "D"  "R2"
[14] "D2" "B'" "L2" "D"  "B'" "R'" "U"  "R2" "B"  "D'" "F'"
```

4 Animations and Flick Books

The `plot` and `plot3D` functions produce 2D and interactive 3D plots of a cube. You can similarly produce 2D and 3D displays of an entire move sequence, which you can turn into flick books and movies (e.g. gifs or mp4 files) respectively. We will continue the example from Section 3.1 of the 4.59 second world record of SeungBeom Cho. The `mv` object is the reconstructed move sequence and `aCube` is the scrambled cube.

4.1 Flick Books

The `move` function by default returns the result of the move sequence `mv` applied to the cube `aCube`. To produce the flick book, we need to call the `move` function with the argument `history` set to `TRUE`, which will then return a list of every cube obtained during the move sequence. The returned list also stores the move sequence as an R attribute. The length of the list is the length of the move sequence plus one, where the one accounts for the original cube. The entire move sequence can then be plotted using the `plot`[16] function.

```
> res.seq <- move(aCube, mv, history = TRUE)
> plot(res.seq)
```

The plot function will produce plots for all cubes by default, but rotations can be skipped by setting the `show.rot` argument to `FALSE`. The plot titles contain the move and the ordinal number of the move within the sequence; rotations do not increase this number as they are not typically counted as moves. If you are using an interface[17] that overwrites previous plots, then you will need `plot(res.seq, ask = TRUE)` to prompt you for the display of the next plot.

The creation of a flick book is similar but we output the plots to a pdf file called `flick.pdf`, which will have one page per move. We also specify the `title` argument to add a title page to the pdf document.

[16] For help on this function use `?plot.seqCubes`.

[17] Personally I use RStudio which does not overwrite previous plots.

```
> res.seq <- move(aCube, mv, history = TRUE)
> pdf("flick.pdf")
> plot(res.seq, title = "SeungBeom Cho\nWorld Record Solve\n4.59")
> dev.off()
```

4.2 Animations

Animations can be produced using the `animate` function. There are a large number of arguments that control the visual display and the number of frames used for each turn or whole cube rotation. Middle slice turns (E, M and S) can also be made. See `?animate` for more details. For an animation you just need to call the function with the starting cube and the move sequence.

```
> animate(aCube, mv)
```

As with the `plot3D` function, the animation is interactive, so you can zoom and spin the cube around with your mouse while it is animating. To produce movies, the animation needs to be recorded, which can be done using the `movie` argument. If you spin the cube around during the animation, this will be recorded in the movie.

```
> animate(aCube, mv, movie = "wrecord")
```

The movie argument should be a character sting. Here we use `"wrecord"`. This will save each frame of the animation as a png file. The filename will be e.g. `wrecord0142.png` for frame number 143. By default, the files[18] will be contained in a the `wrecord` subfolder of the working directory. The subfolder is created if it does not already exist.

The png frames can be used by external utilities (i.e. outside of R) to produce movies. My preference is the powerful command line utility `ffmpeg`. An alternative option is the ImageMagick software suite which performs similar conversions. A simple example would be to use[19] `ffmpeg -i wrecord%04d.png out.mp4` to produce an mp4 movie. You can produce a gif using `ffmpeg -i wrecord%04d.png out.gif` but the file size will be large. The best option to reduce the file size is to restrict the color palette using:

```
ffmpeg -i wrecord%04d.png -vf palettegen palette.png
ffmpeg -i wrecord%04d.png -i palette.png  -lavfi "paletteuse" out.gif
```

The `ffmpeg` command line utility can also adjust the frame rate, add audio, and do many other things.

[18] You can change this using the `dir` argument. The current working directory is given by `getwd()`.

[19] This will not play in Windows Media Player. The best approach is not to use WMP and use VLC instead. If you really want to use WMP, then try `ffmpeg -i wrecord%04d.png -pix.fmt yuv420p out.mp4` instead.

Multiple Animations

In order to produce aminations of several solves, we can use the `read.cubesolve` function and the `animate` function in a loop such as the following.

```
for(i in 5000:5010)
{
  tst <- read.cubesolve(i, warn = TRUE)
  if(grepl("3x3", tst$description)) {
    cat("id =", i, "\n")
    cat(tst$description, "\n\n")
    if(length(tst$scramble) != 0 && length(tst$solution) != 0)
      animate(move(getCubieCube(), tst$scramble), tst$solution)
  }
}
```

The loop is over the id numbers on the cube solves website. The data is read with `read.cubesolve`, and the description is tested for the characters `3x3` by the function `grepl`. The will be true for 3x3, 3x3OH (one-handed) and 3x3BLD (blind) events. The description and id number are printed to the screen, and if the scramble and solution are both present, then the animation is performed. We use `move` to produce the scrambled cube because in blind events the scramble may contain a rotation or a wide move. The `warn = TRUE` option in `read.cubesolve` ensures that the loop continues if the id does not exist.

5 Rotations and Equivalence

The are nine rotation moves given by `x y z x2 y2 z2 x' y'` and `z'`, but there are more than nine spatial orientations of the cube. This is because rotations such as `x'y` cannot be reduced to a single rotation move. Any cube can be rotated[20] in one of 24 ways, including the original cube which corresponds to not performing any rotation.

The function `rotations` performs all 24 rotations of a cube and therefore returns a list of 24 cubes. The first element of the list is the original cube. The output of the `rotations` function can then be passed to the `plot` function[21] to display 2D plots of all 24 rotations. The following code provides an example with a random cube.

```
> rCubes <- rotations(randCube())
> plot(rCubes)
```

The plot function will produce all 24 plots by default. If you are using an interface[22] that overwrites previous plots, then you will need `plot(rCubes, ask = TRUE)` to prompt you for the display of the next plot.

[20] There is also a mathematical operation called a reflection which increases the 24 to 48, but we do not consider reflections here. You can also consider the inverse cube, giving 96 cases. The cases may not all be distinct (accounting for recoloring) due to symmetries.

[21] For help on this function use `?plot.rotCubes`.

[22] Personally I use RStudio which does not overwrite previous plots.

It can also be useful to write the plots to a pdf[23] file. The following code will create a pdf file called `rotations.pdf` with 24 pages, one for each 2D plot.

```
> rCubes <- rotations(randCube())
> pdf("rotations.pdf")
> plot(rCubes)
> dev.off()
```

If one cube can be rotated into another cube, then the cubes can in a rotation sense be considered equivalent. The function `all.equal` can be applied to two cubes to test for rotation equivalence, returning `TRUE` or `FALSE`. A different form of equivalence derives from recoloring. Two cubes are equivalent up to recoloring if swapping the sticker colors of one cube can produce the second cube. You can test this equivalence using the `==` comparison operator. The `==` operator will return `TRUE` if the orientations and permutations are equal, but the spatial orientation vectors can be different. To test for exact equality you can use the R function `identical`, which tests if any two R objects are exactly the same.

If the cube has symmetry properties, then a rotation of the cube may be equivalent to the original up to recoloring, in addition to being rotation equivalent. The following code gives an example of this using the easy checkerboard pattern.

```
> aCube <- getCubieCube("EasyCheckerboard")
> bCube <- move(aCube, "x2")
> all.equal(aCube, bCube)  # rotation equivalent ?

[1] TRUE

> aCube == bCube  # recoloring equivalent ?

[1] TRUE

> identical(aCube, bCube)  # identical ?

[1] FALSE

> identical(aCube, getMovesCube("U2D2R2L2F2B2"))  # identical ?

[1] TRUE

> identical(aCube, invCube(aCube))  # self-inverse ?

[1] TRUE
```

[23] See `?Devices` for information on graphical devices in R

6 Solvers

The function `solver` implements the solvers. By default the Kociemba solver is used. The first argument is the cube to be solved. The target state of the solver does not need to be the solved cube. If the target state is not the solved cube, then it should be given as the second argument. If `A` is the cube to be solved and `B` is the target state, then `invCube(B) %v% A` needs to be solvable, or you will get an error. If the target state `B` is not the solved cube, then it is possible for the solver to work even if both `A` and `B` are unsolvable.

All solvers use look-up tables called pruning tables[24] and move tables. These are silently loaded when a solver is first used, but do not need to be loaded again for subsequent use. A solver will therefore take slightly longer the first time it is called. The solvers are lightweight in the sense that they use small[25] pruning tables, but they are still fairly fast. The look-up tables are hidden objects and cannot be accessed directly.

```
> aCube <- getCubieCube("BlackMamba")
> solver(aCube, divide = TRUE)

 [1] "U"  "D'" "F"  "R2" "F2" "L"  "F2" "L2" "F"  "."  "U'" "R2" "L2"
[14] "F2" "L2" "B2" "U"  "F2" "L2" "U'" "B2" "D2" "B2"

> tCube <- getCubieCube("EasyCheckerboard")
> solver(aCube, tCube, collapse = "-")

[1] "U-D'-F-R2-F2-L-F2-L2-F-U'-B2-U-B2-U-R2-F2-U'-F2-D'-L2-U2-F2"
```

The two examples above demonstrate additional arguments. The `divide` argument adds a period symbol to represent the division between the first and second phases of the solver. Due to the solving method, the eight moves R R' F F' L L' B B' never occur in the second phase. The `collapse` argument returns a character string rather than a character vector, with the argument itself acting as a separator.

An important argument that impacts the behaviour of the solver is the `maxMoves` argument. The solver will always return a solution in `maxMoves` or less. For the Kociemba solver, the default value is 24 moves. If you change it, then it must[26] be an integer with 20 being the smallest possible value and 30 being the largest.

The time taken by the solver can be quite variable. Below are some timings[27] on my laptop in milliseconds using the `microbenchmark` function from the **microbenchmark** package, with 100,000 random cubes. The median solve time is approximately 0.03 seconds, and over 99% of cubes are solved within 0.13 seconds. However if the value of `maxMoves` is

[24] In the context of the IDA* search algorithm these are also called pattern databases or heuristics.

[25] Unless an R package is specified to contain data only, the total size of any (compressed) data cannot exceed 5MB for packages on CRAN.

[26] This is not actually true. If you set the `bound` argument to `FALSE`, then you can override these limits and set `maxMoves` to whatever you want. However you may end up searching for a solution that does not exist, so the solver may be searching until the end of time. Or it may eventually give an error.

[27] The timing includes the cube simulation time, which is about one millisecond.

20 and you hit a difficult case, you could be waiting a while. If you are experimenting with the `maxMoves` argument or you have a case that takes longer than expected, I advise setting the `verbose` argument to `TRUE`, which will allow you to track the progression of the search algorithm.

```
> tp <- microbenchmark(solver(randCube()), times = 100000)
> round(quantile(tp$time/10^6, prob = c(0,.01,seq(.05,.95,.05),.99,1)))
```

```
  0%   1%   5%  10%  15%  20%  25%  30%  35%  40%  45%  50%
   8    9   11   12   13   14   15   17   20   22   25   28
 55%  60%  65%  70%  75%  80%  85%  90%  95%  99% 100%
  32   35   37   39   43   48   57   72   86  125  415
```

The `solver` function has a `type` argument to specify the type of solver. The default is `"KB"`, which is the Kociemba solver. Alternative values[28] are `"ZT"` for the Zemdegs-Twist[29] solver and `"TF"` for the Twist-Flip solver.

In the Zemdegs-Twist solver, the move sequence solves the cube except for corner orientation, and a twist vector (an R attribute for the move sequence) tells you how to twist the corners to finally solve the cube. The Zemdegs-Twist solver can be used by the `scramble` function, in which case the corner twists are performed as the first action of the scramble.

The Twist-Flip solver is similar to the Zemdegs-Twist solver: the move sequence solves the cube except for corner and edge orientation, and twist and flip vectors tell you how to twist the corners and flip the edges to solve the cube. The Twist-Flip solver can also be used by the `scramble` function. The default `maxMoves` value is 20 for Zemdegs-Twist and 16 for Twist-Flip, with a minimum allowable[30] value of 16 for Zemdegs-Twist and 12 for Twist-Flip.

```
> solver(randCube(), type = "ZT")
```

```
 [1] "B"  "L'" "F'" "R'" "D2" "F"  "U'" "F2" "R2" "U'" "R2" "D'" "F2"
[14] "D"  "R2" "L2" "B2" "L2"
attr(,"twist")
URF UFL ULB UBR DFR DLF DBL DRB
  1   1   0   0   2   2   1   2
```

```
> solver(randCube(), type = "TF")
```

```
 [1] "D2" "L'" "B"  "U"  "R2" "F2" "R2" "B2" "L2" "D2" "F2" "B2" "D"
[14] "L2" "F2" "L2"
attr(,"twist")
```

[28] If you read the code you may notice that some of it corresponds to ZZ and CFOP methods and last layer algorithms. My intention was to include pseudo-human solvers for ZZ and CFOP, but they were unfinished when I reached my self-imposed deadline for the package release. I left some of this code in the package for future development. Maybe next year.

[29] Named after a corner twist from Australian speedcuber Felix Zemdegs.

[30] Unless `bound` is `FALSE`.

```
URF UFL ULB UBR DFR DLF DBL DRB
  0   0   2   0   2   1   2   2
attr(,"flip")
FR FL BL BR UR UF UL UB DR DF DL DB
 0  0  0  0  1  0  1  1  0  0  1  0
```

7 Afterword

I hope you get some use from the package: it was written as a hobby project. If you wish to report a bug or submit a feature request for an R package, you need to contact the package maintainer. For the **cubing** package the name and contact for the maintainer is given by `maintainer("cubing")`. This is currently me, but if the past is anything to go by, it may be someone else in the future: I have authored a decent number of R packages but currently maintain only three. Authorship is short-term, but maintenance is a life sentence.

It would be great to hear about your experiences with the package: R users who are also cubers are very rare, so I suspect this will be the least used package that I have ever written. If you are still reading at this point, then I imagine you are one of the few. I will also be far more accommodating to cubers than mathematicians, statisticians or data scientists. But don't tell them that.

Alec Stephenson
Data Scientist, CSIRO
Adj. Assoc. Prof., Swinburne University