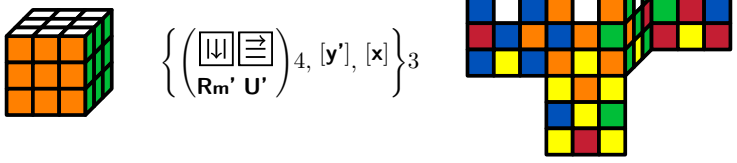
# The RUBIKROTATION package

RWD Nickalls (dick@nickalls.org)
A Syropoulos (asyropoulos@yahoo.com)

This file describes version 5.0 (2018/02/25)
www.ctan.org/pkg/rubik

**Abstract**

The RUBIKROTATION package is a dynamic extension for the RUBIKCUBE package (both are part of the Rubik 'bundle'). This package provides the `\RubikRotation` command which processes a sequence of Rubik rotation moves on-the-fly (using the Perl script `rubikrotation.pl`), and returns the new Rubik cube state (configuration). The RUBIKROTATION package also provides commands for saving the cube state to a file (`\SaveRubikState`), and for displaying any errors (`\ShowErrors`).




## Contents

**1 Introduction** **3**

**2 Requirements** **3**

**3 Installation** **3**
3.1 Generating the files . . . . . 3
3.2 Placing the files . . . . . 4

**4 Usage** **5**
4.1 Enabling the TEX 'shell' facility . . . . . 5
4.2 Configuration-file . . . . . 6

**5 Commands** **7**
5.1 RubikRotation . . . . . 8
5.1.1 Examples . . . . . 9
5.1.2 Sequence strings . . . . . 10

5.1.3 Sequences as macros . . . 11
5.1.4 Arguments in square brackets . . . 12
5.1.5 Groups . . . 13
5.1.6 Random rotations . . . 13
5.2 SaveRubikState . . . 14
5.3 CheckState . . . 14
5.4 ShowErrors . . . 15

**6 Files generated** **15**

**7 General overview** **15**

**8 Change history** **17**

**9 References** **19**

**10 The code** **19**
10.1 Package heading . . . 20
10.2 Some useful commands . . . 20
10.3 Configuration file . . . 21
10.4 Clean file rubikstateNEW.dat . . . 21
10.5 rubikstateERRORS.dat . . . 22
10.6 Setting up file-access for new files . . . 22
10.7 Saving the Rubik state . . . 23
10.8 SaveRubikState command . . . 23
10.9 RubikRotation command . . . 24
10.10 ShowErrors command . . . 25
10.11 CheckState command . . . 25

# 1 Introduction

The RUBIKROTATION package is a dynamic extension to the RUBIKCUBE and RUBIKTWOCUBE packages. It consists of a style option (`rubikrotation.sty`), a Perl script (`rubikrotation.pl`).

The primary role of the RUBIKROTATION package is to implement a sequence of Rubik rotation moves on-the-fly using the `\RubikRotation` command. Consequently the RUBIKROTATION package requires access to the TEX `write18` facility, which is enabled by using the `--shell-escape` command-line switch. This allows command-line control of the Perl script, which is really the 'engine' of this package.

The RUBIKROTATION package has been road-tested on a Microsoft platform (MiKTeX and Strawberry Perl [1]), on a Linux platform (Debian v8.2.0, TEXLive 2017, and Perl v5.20.2), and on a Solaris platform (OpenIndiana).

The following commands are made available by `rubikrotation.sty`:

```
\RubikRotation[]{}
\SaveRubikState
\CheckState
\ShowErrors
\SequenceName
\SequenceInfo
\SequenceShort
\SequenceLong
```

Note that the RUBIKTWOCUBE package makes available the (equivalent) `\TwoRotation[]{}` and `\SaveTwoState` commands.

# 2 Requirements

The RUBIKROTATION package requires the TikZ and the RUBIKCUBE packages.

# 3 Installation

## 3.1 Generating the files

Place the file `rubikrotation.zip` into a temporary directory, and unzip it. This will generate the following files:

```
rubikrotation.ins
rubikrotation.dtx
rubikrotation.pdf     --this document
rubikrotation.pl      --Perl script
rubikrotationPL.pdf   --documentation of rubikrotation.pl
rubikrotation.1       --manual file for rubikrotation.pl ('man' file)
rubikrot-doc-figA.pdf
```

[1] 'Strawberry Perl' (`http://strawberryperl.com`) is a free Perl environment for MS Windows, designed to be as close as possible to the Perl environment of Unix/Linux systems.

```
rubikrot-doc-figB.pdf
rubikrot-doc-figC.pdf
rubikrot-doc-figD.pdf
```

The main package documentation is the file `rubikrotation.pdf`. The documentation of the Perl program `rubikrotation.pl` is the file `rubikrotationPL.pdf`

The style option `rubikrotation.sty` is generated by running (pdf)LaTeX on the file `rubikrotation.ins` as follows:

```
pdflatex  rubikrotation.ins
```

The documentation file (`rubikrotation.pdf`) is then generated using the following sequence of steps [2]:

```
pdflatex    rubikrotation.dtx
pdflatex    rubikrotation.dtx
makeindex -s gind.ist  rubikrotation
makeindex -s gglo.ist -o rubikrotation.gls  rubikrotation.glo
pdflatex    rubikrotation.dtx
pdflatex    rubikrotation.dtx
```

## 3.2 Placing the files

Place the files either in a working directory, or where your system will find them, e.g., in the 'PATH'. The `/texmf-local/` directory tree is often a good place; for example, on a Linux platform with a standard TeX Directory Structure (TDS), then:

*.sty → `/usr/local/texlive/texmf-local/tex/latex/rubik/`
*.cfg → `/usr/local/texlive/texmf-local/tex/latex/rubik/`
*.pdf → `/usr/local/texlive/texmf-local/doc/rubik/`
*.pl → `/usr/local/texlive/texmf-local/scripts/rubik/`

PERL SCRIPT: Make the perl script executable (`chmod +x rubikrotation.pl`), and then rename the file as 'rubikrotation' (i.e., with no file extension), and then place the executable script in the 'PATH', or possibly, directly into your TeXLive binary directory, e.g., `/user/local/texlive/YYYY/bin/i386-linux`.

Sometimes the setting up of a simple one or two-line plain-text configuration-file may be useful or even necessary, depending on your system (see Section 4.2 below). Such a file (if one exists) will automatically be read by `rubikrotation.sty` providing the file is named `rubikrotation.cfg`.

THE 'MAN' FILE: On a Linux platform the manual file (`rubikrotation.1`) is typically located in either `/usr/local/man/man1` or `/usr/local/share/man/man1`. TeXLive typically places such files in the directory `/texmf-dist/doc/man/man1`.

[2] Since the documentation includes a complicated indexing system as well as an index and hyperef links (the package `hypdoc` is used), then a lot of pdflatex runs are required. Prior to the first run it is a good idea to delete any relevant `.toc`, `.aux`, `.out` files.

FILE DATABASE: Finally, (depending on your system) update the TEX file database. For example, on a Linux platform this is achieved using the `texhash` command, or by using the TEXLive Manager (`tlmgr`).

QUICK TEST: To test that your system can now run the perl script, just type at the command-line

```
rubikrotation  -h
```

which should generate something like the following:

```
This is rubikrotation version ...
      Usage: rubikrotation [-h] -i <input file> [-o <out file>]
      where,
      [-h|--help]      gives this help listing
      [-v|--version]   gives version
      [-i]             creates specified input file
      [-o]             creates specified output file
      For documentation see: rubikrotation.pdf,
      rubikrotationPL.pdf and  rubikcube.pdf
```

# 4 Usage

Load the packages `rubikcube.sty`, `rubikrotation.sty`, `rubikpatterns.sty` and `rubiktwocube.sty` in the TEX file preamble *after* loading the TikZ package (all the Rubik packages require the TikZ package). Load the `rubikcube.sty` *before* the other Rubik bundle packages; for example, as follows:

```
\usepackage{tikz}
\usepackage{rubikcube,rubikrotation,rubikpatterns,rubiktwocube}
```

and run (pdf)LATEX using the `--shell-escape` command-line option (see the following section).

## 4.1 Enabling the TEX 'shell' facility

In order to access the TEX `\write18` syntax (so we can access system commands, and hence run the Perl script), it is necessary to invoke the LATEX engine (e.g., (pdf)LATEX or LuaLATEX) using the `--shell-escape` command-line option; for example:

```
pdflatex  --shell-escape  filename.tex
```

In practice, it is probably most convenient to run this command via a bash/batch file. For example, on a Linux platform the following bash file will run the file, show any errors, and open the PDF using AcrobatReader.

```
pdflatex  --shell-escape   filename.tex
echo "...checking error file"
grep ERROR  ./rubikstateERRORS.dat
acroread filename.pdf &
```

If the LaTeX engine is LuaLaTeX, e.g.,

```
lualatex --shell-escape filename.tex
```

then rubikrotation.sty will automatically load the recently developed shellesc package in order to facilitate system access to Perl (see Section 10.1). See *LaTeX News*, issue 24, Feb 2016 for further details of the shellesc package. Consequently, if you intend to use LuaLaTeX then you will need to ensure your system has access to the shellesc package (it can always be downloaded from CTAN directly).

## 4.2 Configuration-file

It is important to realise that the default action of rubikrotation.sty is to access the Perl script as an executable file. This is because the default definitions in rubikrotation.sty are as follows: (they are detailed in Section 10.2)

```
\newcommand{\rubikperlname}{rubikrotation}
\newcommand{\rubikperlcmd}{\rubikperlname\space%
                    -i rubikstate.dat -o rubikstateNEW.dat}
```

Note the need here (in the second macro) to use \space on the end of (\rubikperlname) in order to force a following space—i.e., before the first command-line argument.

If the Perl script has not been made executable, or if you wish to alter how the RUBIKROTATION package accesses the Perl script, then you need to create a plain-text configuration file in order to redefine one or both of the above commands, as described below.

A plain-text configuration-file with the name rubikrotation.cfg (if one exists) will automatically be read by rubikrotation.sty. The RUBIKROTATION package's facility to use a configuration-file allows the user to change not only (a) the filename of the Perl script (rubikrotation.pl), but also (b) the command-line code used by rubikrotation.sty for calling the Perl script. This sort of fine-tuning can be very useful, and sometimes may even be necessary (depending on your system) for running the Perl script.

For example, on some systems it maybe preferable to use a different PATH, file-name and/or a different command-line code to call the script. Such a configuration-file can also facilitate testing a new Perl script having a different name and location.

\rubikperlname
\rubikperlcmd

The configuration-file is essentially a convenient software vehicle for feeding additional LaTeX code to the style option rubikrotation.sty, and hence allows the contents of some commands to be easily adjusted and/or fine-tuned. For the RUBIKROTATION package there are two particular macros we may wish to adjust (see above). The first is that holding the filename of the Perl script, namely \rubikperlname. The second is that holding the command-line call, namely \rubikperlcmd. The following examples illustrate how the configuration-file may be used.

EXAMPLE 1: Suppose we wish to test out a slightly modified Perl script with the working (executable) name rubikrotationR77. In this case we simply create,

in the local working directory, a plain-text configuration-file (it *must* be named exactly `rubikrotation.cfg`) containing just the following line:

```
\renewcommand{\rubikperlname}{rubikrotationR77}
```

EXAMPLE 2: Alternatively, suppose we wish to test out a new Perl script with the (non-executable) name `rubikrotationR55.pl`. Now, in this particular case we will need to run the script using a slightly different command, namely, `perl rubikrotationR55.pl ...`, and consequently we need to implement *both* these changes (of name and command) in the configuration-file, as follows:

```
\renewcommand{\rubikperlname}{rubikrotationR55.pl}
\renewcommand{\rubikperlcmd}{perl  \rubikperlname\space\%
                    -i rubikstate.dat   -o rubikstateNEW.dat}
```

Remember to make sure the PATH associated with the command is also correct.

PLACING THE CONFIGURATION-FILE: The simplest arrangement is just to include the `.cfg` file in the working directory. Alternatively, the `.cfg` file could be placed in the `/texmf-local/` directory tree (say, in `/usr/local/texlive/texmf-local/tex/latex/rubik/`), but in this case one would then have to be careful to specify the correct PATH for everything in order to enable your system to find all the various components etc.

Note that you can, of course, have several `.cfg` files, since the system will read only one such file (the first one it finds starting with the current working directory). Consequently, it may be useful to have one `.cfg` file in your `/texmf-local/` dir (for running the standard Rubik package), and another (different) `.cfg` file in your 'test' directory.

# 5 Commands

The *only* 'Rubik bundle' commands which *must* be used inside a TikZ picture environment are the `\Draw...` commands (these are all provided by the RUBIKCUBE package), although most commands can be placed inside a TikZ environment if you wish.

Using commands which influence the Rubik colour state (e.g., `\RubikFace..`, `\RubikCubeSolvedWY` etc.) outside the `tikzpicture`, `minipage` or `figure` environments generally offers maximum flexibility, since the effects of such commands when used inside these environments remain 'local' to the environment, and are not therefore accessible outside that *particular* environment (see also Section 4.1 in the RUBIKCUBE documentation).

Conversely, the only RUBIKROTATION command which should *not* be used inside a TikZ environment is the `\ShowErrors` command (see the notes on this command below).

## 5.1 \RubikRotation command

\RubikRotation The \RubikRotation[⟨*integer*⟩]{⟨*comma-separated sequence*⟩} command processes a sequence of rotation codes, and returns the final state. The inverse sequence can also be implemented (see **Inverse** below). Note that the equivalent \TwoRotation command (see the RUBIKTWOCUBE package) behaves in the same way as the \RubikRotation command in all respects.

The first (optional) argument [⟨*integer*⟩] of the \RubikRotation command is the number of times the whole command itself should be repeated; for example as follows: \RubikRotation[2]{...}.

The second (mandatory) argument consists of a comma-separated sequence of rotation codes, e.g., U, D2, which may be encoded as a macro. In addition, there may be additional comma-separated macros and optional [name], 'repeat blocks' and 'info blocks' (see below). The general structure of the second argument is as follows: \RubikRotation{[name],...,\macro,...,(repeat)n,...,<info>}. These elements are now described in detail.

**Square brackets**: This is an optional 'sequence name' facility. The contents of square brackets are not processed as rotations, and can therefore be used as a 'name' of the sequence, e.g., [CrossSeq], or as a tag, to be visible in the log file. The contents must *not* include commas, but can have other separators, e.g., spaces, semicolons etc. Importantly, the contents of the first square bracket will be designated the sequence name and will be held as the macro \SequenceName. Square brackets can also be used in repeat blocks (see below). Square brackets must be separated by a comma from adjacent codes.

**Repeat block**: This is an optional comma-separated sequence of rotation codes which is to be repeated a specified number of times. It must be delimited by balanced curved brackets, and an optional terminal integer $n$ (repeat number) can be used. For example, (F,B3)2, where the '2' indicates that the rotation sequence F,B3 is to be processed twice. If the repeat number is omitted then $n = 1$ is assumed. Repeat blocks must be separated by a comma from adjacent codes, and can include balanced square brackets (see below).

**Info block**: This is an optional block of meta information, and must be delimited by balanced angle-brackets <..>. An info-block typically carries information regarding the sequence itself; typically, something like <(20f*) //C2(a)>. If an infoblock includes the keyword 'inverse' then the program will implement the inverse sequence of rotations (see below). An info-block must be separated by a comma from the adjacent codes. The contents of all info blocks will be held collectively as the macro \SequenceInfo.

**Inverse sequence**: The (mathematically) inverse sequence of the given sequence can be implemented by including the keyword 'inverse' (or INVERSE) in an infoblock, as follows \RubikRotation{<inverse>, ... }. The keyword can be either in its own separate infoblock, or inside a larger infoblock. The program simply checks for the string 'inverse', which can be either lower-case or upper-case. The implemented sequence can be checked by looking at (or printing) the contents of the macro \SequenceLong (see section on *Sequence strings* below). Note that the macro \SequenceLong is also shown (expanded) in the log file.

### 5.1.1 Examples

Some examples of the use of the \RubikRotation command are as follows; the commas are important and brackets must be balanced and not nested:

```
\RubikRotation[2]{x,R2,U}
\RubikRotation{\sixspot}
\RubikRotation{<inverse>,[myseqB],U,D,L,R2,(M,U)3,D2}
\RubikRotation{[K32466],U,F,U2,F,L2,B,U2,F,Lp,Rp,F2,D,R2,U2,L2,B,Fp,
 L,F2,D,<(20f*) //Oh{I}>}
```

**Inverse sequence**

Inverting a sequence involves (a) reversing the order, and (b) inverting each element. Thus, the inverse of the sequence (Up,D,L2,Rp) is (R,Lp,Lp,Dp,U). But (Lp,Lp) ≡ L2, and so the inverse of (Up,D,L2,Rp) would generally be expressed as (R,L2,Dp,U). However, since the macro \SequenceLong records the individual elements as they are processed, when a sequence is inverted notational compressions such as Lp,Lp → L2 are not made. For example, processing the command \RubikRotation{<inverse>,Up,D,L2,Rp} results in the macro \SequenceLong being displayed in the subsequent rubikstateNEW.dat file as

```
\renewcommand\SequenceLong{R,Lp,Lp,Dp,U}%
```

A more extensive example is given at the end of Section 5.1.2.

**Repetitions**

Repetitions can be achieved in various ways. First, all the rotations in the second argument can be repeated multiple times, say $n$ times, by using the optional [n] argument of the \RubikRotation[]{} command; i.e., the whole of the mandatory argument of the \RubikRotation command is then executed $n$ times.

Second, a sub-sequence of rotations can be repeated within the main argument multiple times, by delimiting such groups with curved brackets and a trailing integer (i.e., in a repeat-block), as described above. If no integer is given, then $n = 1$ is assumed, and hence curved brackets can also be used simply to highlight particular sequences. For example, the following five commands are equivalent:

```
\RubikRotation[3]{x,R2,U}
\RubikRotation{(x,R2,U)3}
\RubikRotation{(x,R2,U)2,x,R2,U}
\RubikRotation{x,R2,U,x,R2,U,x,R2,U}
\RubikRotation{(x,R2,U),(x,R2,U),(x,R2,U)}
```

**Macros**

Note also that macros representing a rotation sequence can also appear as part of the main argument. So, extending the previous example, if we were to define \newcommand{\ShortSeq}{x,R2,U}, then the following three commands would also be equivalent to the five previous ones:

```
\RubikRotation[3]{\ShortSeq}
\RubikRotation{(\ShortSeq)3}
\RubikRotation{(x,R2,U),\ShortSeq,\ShortSeq}
```

**Process overview**

The `\RubikRotation` command results in LaTeX first writing the current Rubik state to a text file (`rubikstate.dat`), and then calling the Perl script `rubikrotation.pl`. The Perl script then reads the current Rubik state from the (`rubikstate.dat`) file, performs all the rotations, and then writes the new Rubik state, and the four `\Sequence...` macros (see below), and any error messages, all to the file `rubikstateNEW.dat`, which is then input on-the-fly by the `.tex` file. This new Rubik state can then be used either as the input for another `\RubikRotation` command, or used to generate a graphic image of the cube. The `\Sequence...` macros can then be used for typesetting the sequence of rotations in various formats.

### 5.1.2 Sequence strings

`\SequenceName` `\SequenceInfo` `\SequenceShort` `\SequenceLong`

The sequence of rotation codes used as the main argument for the `\RubikRotation` command is also returned in the form of four macros, namely `\SequenceName` (contains the 'name' of the sequence if a `[name]` exists), `\SequenceInfo` (contains any sequence meta data in 'info-blocks'), `\SequenceShort` (contains the original sequence of rotation codes), and `\SequenceLong` (contains the expanded or 'Long' form of the original sequence—i.e., in which any 'short' rotation codes (e.g., `R2, L3`) in the original sequence have been expanded into their separate codes—e.g., `R, R, L, L, L` etc.).

For example, if we wanted to see the effect of the sequence associated with the 'SixTs' cube configuration `[SixTs],F2,R2,U2,Fp,B,D2,L2,F,B,<(14q*,14f*)>` on a solved Rubik cube (where 'SixTs' is the 'name' of the sequence), we could use the following commands:

```
\RubikCubeSolved   % sets up the colours for a solved cube state
\RubikRotation{[SixTs],F2,R2,U2,Fp,B,D2,L2,F,B,<(14q*,14f*)>}
\ShowCube{2.8cm}{0.7}{\DrawRubikCubeRU}
```

Note (a) contents of a square bracket [..] are not processed as rotations, (b) the contents of the first square bracket in a sequence is taken to be the 'name' of the sequence (see Section 5.1.4 for more details). In this example the four `\Sequence..` macros described above would now hold the following strings:

`\SequenceName` = SixTs
`\SequenceInfo` = (14q*,14f*)
`\SequenceShort` = [SixTs],F2,R2,U2,Fp,B,D2,L2,F,B
`\SequenceLong` = F,F,R,R,U,U,Fp,B,D,D,L,L,F,B

As another example, we now show how to implement the inverse of the above `SixTs` sequence, by including the key word 'inverse' in an infoblock, and, more conveniently, using the macro `\sixts` from the RUBIKPATTERNS package, as follows:

```
\RubikRotation{<inverse>,\sixts}
```

In this case, the log file would then show the associated `\Sequence..` macros as follows:

```
...SequenceName = SixTs
...SequenceInfo = inverse; (14q*; 9f*)
...SequenceShort = [SixTs],F2,R2,U2,Fp,B,D2,L2,F,B
...SequenceLong = Bp,Fp,Lp,Lp,Dp,Dp,Bp,F,Up,Up,Rp,Rp,Fp,Fp
```

showing that the macro `\SequenceShort` holds the `\sixts` sequence, while the macro `\SequenceLong` holds the inverse sequence which was actually implemented.

For further details regarding the use of these `\Sequence..` macros for typesetting the various components of a sequence, and why the `\SequenceLong` command is particularly useful, see Section 10 in the RUBIKCUBE documentation (the `\ShowSequence` command).

### 5.1.3 Sequences as macros

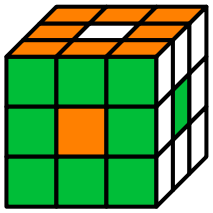
Macros which are arguments of the TEX `\write` command are expanded on writing (Eijkhout 1992, § 30.2.3, p. 238)[see references Section 9]. Consequently we are able to use a sequence-defining macro as an argument for the `\RubikRotation` command. In fact this is very convenient, since it allows one to store lots of different rotation sequences by name alone. Note that `rubikpatterns.sty` (part of the Rubik bundle) is a collection/database of many such well-known named sequences.

For example, by installing the RUBIKPATTERNS package we are able to use the name 'sixspot' for a macro denoting the rotation sequence which generates the well known 'sixspot' configuration (see the 'patterns' page on the Reid website)[see references Section 9]. The 'sixspot' sequence is defined as follows:

```
\newcommand{\sixspot}{U,Dp,R,Lp,F,Bp,U,Dp,<(8q*, 8f*)>}
```

Armed with the `\sixspot` macro we are now able to generate the graphic (sixspot cube) very easily using the following code—this time we demonstrate the use of the more convenient `\ShowCube` command (which includes the `tikzpicture` environment):

```
\usepackage{rubikcube,rubikrotation}
\usepackage{rubikpatterns}
...
\RubikCubeSolved
\RubikRotation{\sixspot}
\ShowCube{3cm}{0.7}{\DrawRubikCubeRU}
```



Providing such macros (when used as arguments) are comma separated (as the rotation codes must be), then the `\RubikRotation` command can accommodate both rotation codes and macros; for example, `\RubikRotation{x,y,\sixspot,x}`.

### 5.1.4 Arguments in square brackets

The contents of a square bracket are not processed as rotations, but are simply interpreted as an inactive 'string'. This feature therefore allows the contents to be used as a label, which can be very useful. Note the contents of square brackets must not include commas, but spaces and semicolons are allowed.

For example, we can use this facility to 'name' the 'SixSpot' configuration mentioned above, as follows:

```
\RubikRotation{[SixSpot],U,Dp,R,Lp,F,Bp,U,Dp}
```

In practice, it is quite useful to go one step further and include the [] label-name feature in the `\sixspot` command, as follows,

```
\newcommand{\sixspot}{[SixSpot],U,Dp,R,Lp,F,Bp,U,Dp}
```

Note that using the `[name]` facility has the great advantage of making the label-name visible in the log-file. For example, the following command, which uses the rotations **x2**, and **y** to rotate the Rubik cube after applying the 'sixspot' sequence of rotations:

```
\RubikRotation{\sixspot,x2,y}
```

will then be represented in the log file as

```
...dataline = rotation,[SixSpot],U,Dp,R,Lp,F,Bp,U,Dp,<(8q*; 8f*)>,x2,y
...[SixSpot] is a label OK
...rotation U, OK
...rotation Dp, OK
...rotation R, OK
...rotation Lp, OK
...rotation F, OK
...rotation Bp, OK
...rotation U, OK
...rotation Dp, OK
...Expanding x2 ...
...rotation x, OK (= x = R + Sr + Lp)
...rotation x, OK (= x = R + Sr + Lp)
...rotation y, OK (= y = U + Su + Dp)
...writing new Rubik state to file rubikstateNEW.dat
...SequenceName = SixSpot
...SequenceInfo = (8q*; 8f*)
...SequenceShort = [SixSpot],U,Dp,R,Lp,F,Bp,U,Dp,x2,y
...SequenceLong = U,Dp,R,Lp,F,Bp,U,Dp,x,x,y
```

Note that the `\sixspot` macro, as held in the RUBIKPATTERNS package, includes a terminal infoblock holding the 'SequenceInfo' as indicated in the above example.

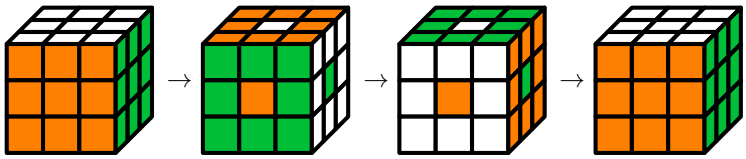
Also, note that the square bracket feature allows for several named rotation sequences to be easily distinguished in the log file from adjacent rotation sequences. This feature is also useful when typesetting a sequence of rotation codes, since the first element will then appear in the form `[name]`, obviating the need to typeset the name of the sequence separately.

See also the `\ShowSequence` command (in the RUBIKCUBE package) for a convenient way of displaying a sequence of rotations in various formats.

### 5.1.5 Groups

The `\RubikRotation` command is a convenient tool for illustrating how Rubik rotations and sequences of rotations are elements of groups and subgroups. For example, using the RUBIKROTATION package it is easy to show that three cycles of the 'sixspot' sequence return the Rubik cube to its original state. More formally this is equivalent to $(\texttt{\textbackslash sixspot})3 \equiv e$ [3], and can be nicely illustrated by implementing the following pseudocode:

```
\RubikCubeSolved . \RubikRotation[3]{\sixspot} = \RubikCubeSolved
```



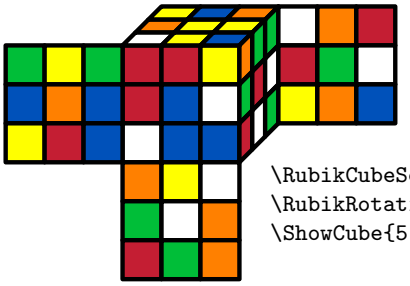
### 5.1.6 Random rotations

The `\RubikRotation` command can also be used to scramble the cube using a random sequence of rotations. If the first argument is the word 'random' AND the second argument is an integer $n$, $(1 \leq n \leq 200)$, then a random sequence of $n$ rotations will be performed. If $n > 200$ then the currently set maximum value $n = 200$ will be used.

As a safety feature the maximum $n$ can be changed only by editing the set value of the Perl variable `$maxn` in the Perl script `rubikrotation.pl`, where we currently have (see the 'random' subroutine in the document `rubikrotationPL.pdf`)

```
my $maxn=200;
```

For example, the following commands scramble a solved cube using a sequence of 120 random rotations, and display the state in the form of a semi-flat (SF) cube.



```
\RubikCubeSolved%
\RubikRotation{random,120}%
\ShowCube{5.5cm}{0.5}{\DrawRubikCubeSF}
```

Note that in this particular example (above), only the `\Draw..` command is inside the TikZ picture environment (i.e., inside the `\ShowCube` command). Note

[3] $e$ is the 'identity' element

also that when Rubik commands are outside a TikZ picture environment, they should have a trailing % to stop additional white space being included.

The randomisation procedure is as follows: all the possible rotations are first allocated a different cardinal number (positive integer) and collected into an array. Then a sequence of $n$ randomised numbers is generated and mapped to the array to generate the associated sequence of random rotations. The sequence used is detailed in the `.log` file.

## 5.2 \SaveRubikState command

`\SaveRubikState` The command `\SaveRubikState{`⟨*filename*⟩`}` saves the state (configuration) of the Rubik cube to the file `{`⟨*filename*⟩`}` in the standard `\RubikFace...` format so that it can be read by LaTeX. Consequently such a file can then be input so it can be drawn or processed in the usual way. The output file is 'closed' immediately following the 'write' in order to allow it to be available for input later by the same file if required.

For example, the following commands would save the so-called 'sixspot' configuration (generated by the rotations **U, Dp, R, Lp, F, Bp, U, Dp**) to the file `sixspot.tex`.

```
\RubikCubeSolved%
\RubikRotation{U,Dp,R,Lp,F,Bp,U,Dp}%
\SaveRubikState{sixspot.tex}%
```

The form of the file `sixspot.tex` will then be as follows—the filename (commented out) is automatically written to the top of the file for convenience.

```
% filename: sixspot.tex
\RubikFaceUp{O}{O}{O}{O}{W}{O}{O}{O}{O}%
\RubikFaceDown{R}{R}{R}{R}{Y}{R}{R}{R}{R}%
\RubikFaceLeft{Y}{Y}{Y}{Y}{B}{Y}{Y}{Y}{Y}%
\RubikFaceRight{W}{W}{W}{W}{G}{W}{W}{W}{W}%
\RubikFaceFront{G}{G}{G}{G}{O}{G}{G}{G}{G}%
\RubikFaceBack{B}{B}{B}{B}{R}{B}{B}{B}{B}%
```

We can therefore access and draw this configuration in a Semi-Flat format later, when required, simply by inputting the file as follows:

```
\input{sixspot.tex}
\ShowCube{7cm}{0.7}{\DrawRubikCubeSF}
```

## 5.3 \CheckState command

`\CheckState` Since it is easy to inadvertently define an invalid Rubik cube (e.g., enter an invalid number of, say, yellow facelets), this command simply checks the current colour state of all the cubies of a 3x3x3 Rubik cube, and shows the number of facelets of each colour. An ERROR: code is issued if the number of facelets having a given colour exceeds 9 for a 3x3x3 cube (Rubik cube), or exceeds 4 for a 2x2x2 cube

(Two cube). The results are written to the the .log file, and displayed at the point the \ShowErrors command is used (for code see Section 10.11).

One can check the current Rubik state (for errors) by issuing the command

```
\CheckState%
```

Note (1) that such a check is implemented automatically with each \RubikRotation and \TwoRotation command, and (2) and makes only a very superficial check—simply counting the number of cubies of each colour.

Note that the \CheckState command replaces the earlier \CheckRubikState, but we retain the old command for backwards compatibility (for the moment).

### 5.4 \ShowErrors command

\ShowErrors

Any errors which arise can be made visible using the command \ShowErrors. This command places a copy of the 'error' file (rubikstateERRORS.dat) underneath the graphic image so you can see any errors if there are any (note that details of all errors are are written by default to the .log file) (for code see Section 10.10).

Consequently, the \ShowErrors command must be placed *after* a TikZ picture environment—it cannot be used inside a TikZ environment. In fact this command is probably best placed at the end of the document where it will reveal all rotation errors generated while processing the whole document. Once the document is free of errors this command can be removed or just commented out. The file rubikexamples.pdf shows an example of the use of this command.

Note that the \ShowErrors command replaces the original \ShowRubikErrors command, which will be retained for backward compatibility.

## 6 Files generated

Whenever the \RubikRotation or \CheckRubikState commands are used, three small temporary plain-text files for holding data are generated as follows (they are refreshed with each LaTeX run, and are not actively deleted).

- LaTeX writes Rubik state data to the file rubikstate.dat.
- The Perl script rubikrotation.pl reads the file rubikstate.dat and then writes the new rubik state to the file rubikstateNEW.dat.
- The Perl script rubikrotation.pl also writes error data to the file rubikstateERRORS.dat. A copy of this file is displayed under the graphic image when the command \ShowErrors is used after the TikZ picture environment.

## 7 General overview

When LaTeX processes rubikrotation.sty the following steps are implemented (see Section 10):

1. A check is made to see if `fancyvrb.sty` is loaded: if not then this package is loaded (this package supplies the command `\VerbatimInput` which is required for inputting the file `rubikstateERRORS.dat` in verbatim form).
2. A check is made to see if a configuration-file (`rubikrotation.cfg`) exists: if so then this file is input.
3. The text file `rubikstateNEW.dat` is overwritten (if it exists), otherwise the file is created (this prevents an 'old' version of the file being used by LaTeX).
4. The plain-text file `rubikstateERRORS.dat` is created. This file collects error messages generated by the Perl script.

When a `\RubikRotation` command is processed (see Section 10.9, line 102), it first writes the current colour configuration of each face (the 'rubik state') to the temporary file `rubikstate.dat` (this will be read by the Perl script `rubikrotation.pl`). The `\RubikRotation` command also appends the keyword '`checkrubik`' as well as a copy of the string of Rubik rotations. It then calls the Perl script `rubikrotation.pl`.

For example, if we use the command `\RubikCubeSolved` followed by the command `\RubikRotation[2]{U,D,L,R}`, then the associated `rubikstate.dat` file will be written as follows:

```
% filename: rubikstate.dat
cubesize,three
up,W,W,W,W,W,W,W,W,W
down,Y,Y,Y,Y,Y,Y,Y,Y,Y
left,B,B,B,B,B,B,B,B,B
right,G,G,G,G,G,G,G,G,G
front,O,O,O,O,O,O,O,O,O
back,R,R,R,R,R,R,R,R,R
checkstate
rotation,U,D,L,R
rotation,U,D,L,R
```

Note that the `\RubikRotation` option `[2]` results in the string "`rotation,U,D,L,R`" being written twice to the `rubikstate.dat` file, as shown above.

Alternatively, if we used the command `\RubikRotation{random, 45}` then the last line written to the file would be the string "`rotation,random,45`", as follows:

```
% filename: rubikstate.dat
cubesize,three
up,W,W,W,W,W,W,W,W,W
down,Y,Y,Y,Y,Y,Y,Y,Y,Y
left,B,B,B,B,B,B,B,B,B
right,G,G,G,G,G,G,G,G,G
front,O,O,O,O,O,O,O,O,O
back,R,R,R,R,R,R,R,R,R
checkstate
rotation,random,45
```

A \CheckState command triggers the same sequence of events except that no "rotation,...." line is written.

The action of the Perl script rubikrotation.pl is controlled by the keywords (first argument of each line) associated with each line of the file rubikstate.dat. When control passes from LaTeX to Perl, the script rubikrotation.pl starts by loading the current rubikstate (prompted by the keywords 'up', 'down', 'left', 'right', 'front', 'back'), and performing a syntax check—significant syntax errors at this stage will cause the program to issue appropriate error messages and then terminate cleanly. Next, the Perl script performs some basic cube checks (prompted by the key word 'checkstate'), and then the program processes the sequence of Rubik rotations (prompted by the keyword 'rotation'). If, instead, the second argument of the 'rotation,...' string is the keyword 'random', and provided this is followed by a valid integer, say $n$, then the Perl script performs a sequence of $n$ random rotations. Finally, the Perl script writes the final rubikstate to the text file rubikstateNEW.dat. All error messages are written to the text file rubikstateERRORS.dat and also to the LaTeX log-file. The Perl script now closes all open files and terminates.

Control then passes back to LaTeX (still in rubikrotation.sty processing the \RubikRotation command—see Section 10.9, line 115); its next action is to input the file rubikstateNEW.dat. If there are more \RubikRotation commands (in the .tex file) then this cycle repeats accordingly. Eventually a \Draw... command is reached (in the .tex file) and the final rubikstate is drawn in a TikZ picture environment.

If the TikZ picture environment is followed by a \ShowErrors command, then a 'verbatim' copy of the rubikstateERRORS.dat file is displayed immediately under the graphic. Once the graphic is error-free, then the \ShowErrors command can be removed or commented out.

Alternatively, when processing a long document, it can be useful to place a \ShowErrors command at the end of the document, where it will list all errors which occurred. Once any errors have been fixed, this command can be removed or commented out.

Note that if a BASH file is used to coordinate the process then it is often convenient to use the Linux grep utility to alert the user to any run-time errors, by using grep to scan the rubikstateERRORS.dat file at the end of the run; for example, as follows:

```
pdflatex  --shell-escape  myfile.tex
echo "...checking error file"
grep ERROR  ./rubikstateERRORS.dat
```

# 8 Change history

- Version 5.0 (February 2018)

  — minor bugfixes and better syntax checking in the Perl program. The cube size being processed (ie 3x3x3 or 2x2x2) is now detected by the program.

— the command \CheckState replaces the earlier \CheckRubikState command in order to avoid confusion, now that we are able to process both the 3x3x3 cube (Rubik cube) and also the 2x2x2 cube (Two cube) (see Sections 5.3 and 10.11 (code)). The original command is retained (for now) for backward compatibility. \CheckRubikState → \CheckState

— the command \ShowErrors replaces the earlier \ShowRubikErrors command in order to avoid confusion now, that we are able to process both the 3x3x3 cube (Rubik cube) and also the 2x2x2 cube (Two cube) (see Sections 5.4 and 10.10 (code)). The original command is retained (for now) for backward compatibility. \ShowRubikErrors → \ShowErrors

- Version 4.0 (March 2017)

  — The \RubikRotation command has been enhanced to allow its argument to include so-called repeat-blocks (rotation-sequences which can be repeated multiple times), and info-blocks for holding sequence metadata (see Section 5.1). Syntax checking of the \RubikRotation argument is much improved (see rubikrotationPL.pdf for details of the Perl script).

  — Four new macros which hold derived data (see Section 5.1).

  ```
  \SequenceName
  \SequenceInfo
  \SequenceShort
  \SequenceLong
  ```

  — Better syntax checking of the \RubikRotation argument by the Perl program rubikrotation.pl.

- Version 3.0 (25 September 2015)

  — The \RubikRotation command now actions multiple instances of its argument as determined by an optional 'repeat' [⟨*integer*⟩]. For example the command \RubikRotation[3]{R,x} is equivalent to the command \RubikRotation{R,x,R,x,R,x} (see Sections 5.1 and 10.9).

  — If a comma separated element used as an argument for the \RubikRotation command is prefixed with either a * or [ or ] character then it is not actioned as a rotation (see Section 5.1.4).

  — The Perl script rubikrotation.pl now has command-line switches, including -h to show some 'help' and 'usage' information (see Section 3.2).

  — A 'man' file (manual file) for the Perl script rubikrotation.pl is now included in the package.

  — The Perl script rubikrotation.pl now uses as input and output filenames those specified in the command-line of the CALLing program. This now allows the script rubikrotation.pl to be used as a stand-alone tool (see the rubikrotation 'man' file for details).

  — The documentation for the Perl script rubikrotation.pl is in the accompanying file rubikrotationPL.pdf.

— Fixed typos, index and minor errors in the documentation.

- Version 2.0 (5 February, 2014)
  — First release.

# 9 References


- Abrahams PW, Berry K and Hargreaves KA (1990). *TEX for the impatient* (Addison-Wesley Publishing Company), page 292.
  Available from: http://www.ctan.org/pkg/impatient
  [re: \rubikpercentchar and \@comment in Section 10.2]
- Eijkhout V (1992). *TEX by topic: a TEXnician's reference.* (Addison-Wesley Publishing Company), pages 232 & 238.
  Available from: https://bitbucket.org/VictorEijkhout/tex-by-topic/
  [re: \string in Section 10.8] [re: \write in Section 5.1.3]
- Feuersänger C (2015). Notes on programming in TEX.
  (revision: 1.12.1-32-gc90572c; 2015/07/29)
  http://pgfplots.sourceforge.net/TeX-programming-notes.pdf
  [re: loop macros in Section 10.9]
- Kociemba website (Kociemba H). http://www.kociemba.org/cube.htm
- Nickalls RWD and Syropoulos A (2015). The RUBIKCUBE package, v3.0. http://www.ctan.org/pkg/rubik,
- Randelshofer website (Randelshofer W). http://www.randelshofer.ch/rubik/ [re: sequences and supersetENG notation]
- Reid website (Reid M). Patterns. http://www.cflmath.com/Rubik/patterns.html [re: sequences as macros; in Section 5.1.3]
- Tellechea C and Segletes SB (2016). The listofitems package, v1.2
  http://www.ctan.org/pkg/listofitems


# 10 The code (rubikrotation.sty)

In the following, the term 'Perl script' denotes the script rubikrotation.pl. Useful information regarding the TEX \write command is given in Eijkhout (1992), § 30.2.3 (page 238). For the means of including a '%' character in the token list of \write see Abrahams *et. al* (1990).

## 10.1 Package heading

```
⟨*rubikrotation⟩
\def\RRfileversion{5.0}%
\def\RRfiledate{2018/02/25}%  25 February 2018
\NeedsTeXFormat{LaTeX2e}
\ProvidesPackage{rubikrotation}[\RRfiledate\space (v\RRfileversion)]
```

The package requires `rubikcube.sty`. However `rubikcube.sty` is not automatically loaded (for the moment at least) since this makes it difficult to errorcheck new versions.

```
\@ifpackageloaded{rubikcube}{}{%
   \typeout{---rubikrotation requires the rubikcube package.}%
   }%
```

The RUBIKROTATION package requires access to the fancyvrb package for the `\VerbatimInput{}` command which we use for inputting and displaying the error file (see Section 5.4).

```
\@ifpackageloaded{fancyvrb}{}{%
   \typeout{---rubikrotation requires the fancyvrb package%
       for VerbatimInput{} command.}%
   \RequirePackage{fancyvrb}}
```

For the `\write18` syntax to work with LuaTEX (so we can access system commands) we require the recent shellesc package, which we load using the ifluatex conditional (see Section 4.1).

```
\@ifpackageloaded{ifluatex}{}{%
   \typeout{---rubikrotation requires the ifluatex package.}%
   \RequirePackage{ifluatex}}
\ifluatex%
   \@ifpackageloaded{shellesc}{}{%
   \typeout{---rubikrotation requires the shellesc package
       to run using Lua\LaTeX.}%
   \RequirePackage{shellesc}}
\fi%
```

## 10.2 Some useful commands

`\rubikrotation` First we create a suitable logo

```
\newcommand{\rubikrotation}{\textsc{rubikrotation}}
\newcommand{\Rubikrotation}{\textsc{Rubikrotation}}
```

`\@print` We need a simple print command to facilitate writing output to a file.

```
\newcommand{\@print}[1]{\immediate\write\outfile{#1}}
```

`\@comment` `\@commentone` We also require access to the '%' character so we can (a) write comments to files, including the log file, and (b) place a trailing '%' in a line of code written to a file.

To achieve this we define the '%' character as `\rubikpercentchar` (modified from: Abrahams PW, Berry K and Hargreaves KA (1990), p 292) [see refs Section 9], and also two 'comment' commands which implement it. This 'workaround'

is necessary because TEX does not allow the use of the \% command for placing a '%' character in the token list of \write. See Abrahams *et. al* (1990) for details.

```
{\catcode`\%=12    \global\def\rubikpercentchar{%}}%
\newcommand{\@comment}{\rubikpercentchar\rubikpercentchar\space}%
\newcommand{\@commentone}{\rubikpercentchar}%
```

\rubikperlname This holds the name of the Perl script. A configuration-file (rubikrotation.cfg) can be used to change the default name of the Perl script using a renewcommand (see Section 4.2).

```
\newcommand{\rubikperlname}{rubikrotation}
```

Note that here we are assuming that the script is an executable file.

\rubikperlcmd This holds the command-line code for calling the Perl script Note that the command-line requires a mandatory input filename preceded by the -i switch. An optional output filename (preceded by the -o switch) may be used, otherwise the default output filename of rubik-OUT.dat will be used.

Note that it is very important that we do actually specify an output filename (for receiving data from the Perl script). This is because (a) The Perl script rubikrotation.pl is currently configured to read its output filename as an argument from the command-line (so it can be flexibly used as a stand-alone script for processing a given Rubik state through a sequence of rotations), and (b) rubikrotation.sty is currently configured to read its input (i.e., data generated by the Perl script) from the file rubikstateNEW.dat.

```
\newcommand{\rubikperlcmd}{\rubikperlname\space%
                    -i rubikstate.dat   -o rubikstateNEW.dat}
```

Remember to use the \space macro following the \rubikperlname macro in order to generate the mandatory space before the first command-line argument.

A plain-text configuration-file rubikrotation.cfg can be used to change the default command-line code using a renewcommand (see Section 4.2).

## 10.3 Configuration file

If a configuration file exists (rubikrotation.cfg) then input it here, i.e., *after* defining the \rubikperlname and \rubikperlcmd macros and *before* creating the rubikstateERRORS.dat file.

```
\typeout{---checking for config file (rubikrotation.cfg)...}
\IfFileExists{rubikrotation.cfg}{%
   \input{rubikrotation.cfg}%
   }{\typeout{---no config file available}%
}%
```

## 10.4 Clean file rubikstateNEW.dat

We need to clean out any existing (old) rubikstateNEW.dat file, since if the TeX shell command-line switch is accidentally not used then the Perl script

rubikrotation.pl will not be CALLed, and hence this file will not be renewed (i.e., an 'old' image may be used).

```
\typeout{---cleaning file rubikstateNEW.dat}%
\newwrite\outfile%
\immediate\openout\outfile=rubikstateNEW.dat%
\@print{\@comment rubikstateNEW.dat}%
\immediate\closeout\outfile%
```

## 10.5 rubikstateERRORS.dat

We first open the file rubikstateERRORS.dat which is used by the Perl script rubikrotation.pl for writing its error-messages to. This file is displayed by the command \ShowErrors.

IMPORTANT NOTE: this file is created fresh each time LaTeX is run, and hence the Perl script only appends data to it during the LaTeX run, so this file just grows until either it is destroyed or recreated—this is a useful feature since the file accumulates all error messages as the .tex file is processed. We can't make the Perl script create the file since the Perl script is only CALLed if we use a \RubikRotation or \CheckRubikState command (which we may not!)—so it has to be created here.

```
\typeout{---creating file rubikstateERRORS.dat}%
\newwrite\outfile%
\immediate\openout\outfile=rubikstateERRORS.dat%
\@print{\@comment ShowErrors (rubikstateERRORS.dat)}%
\@print{\@comment ---------------------------------}%
\immediate\closeout\outfile%
```

## 10.6 Setting up file-access for new files

Having set up all the primary files, we now need to set up a newwrite for all subsequent file openings (e.g., for rubikstate.dat and saving to arbitrary filenames by the \SaveRubikState command). Otherwise, we can easily exceed the LaTeX limit of 15. From here-on TeX will use openout7 when opening and writing to files. We will implement new openings using the command \@openstatefile (see below).

```
\typeout{---setting up newwrite for rubikrotation.sty to use...}%
\newwrite\outfile%
```

\@openstatefile
\@closestatefile

We also need commands for easy file opening and closing for new instances of the file rubikstate.dat etc. Note that for this we are therefore using the same outfile number as set up by the \newwrite... above.

```
\newcommand{\@openstatefile}{\immediate\openout\outfile=rubikstate.dat}
\newcommand{\@closestatefile}{\immediate\closeout\outfile}
```

## 10.7 Saving the Rubik state

\@printrubikstate This internal command writes the Rubik configuration to the file `rubikstate.dat`, and is used by the `\RubikRotation` command (see Sections 5.2 and 7). The file `rubikstate.dat` is read by the Perl script, and represents the state on which the new `\RubikRotation` command acts. Note that we include the line `cubesize,three` to inform the Perl script that the cube is a 3x3x3 cube (this is used in the 'random' subroutine).

The actual state (colour state) is simply an ordered sequence of the faces and the colours associated with each facelet of a face. The colour associated with a particular facelet is held by the variable for that facelet. For example, the top-left facelet associated with the FRONT face is held in the variable `\Flt` (see Section 5.2). Further relevant documentation is in the RUBIKCUBE package.

```
\newcommand{\@printrubikstate}{%
   \@print{cubesize,three}%
   \@print{up,\Ult,\Umt,\Urt,\Ulm,\Umm,\Urm,\Ulb,\Umb,\Urb}%
   \@print{down,\Dlt,\Dmt,\Drt,\Dlm,\Dmm,\Drm,\Dlb,\Dmb,\Drb}%
   \@print{left,\Llt,\Lmt,\Lrt,\Llm,\Lmm,\Lrm,\Llb,\Lmb,\Lrb}%
   \@print{right,\Rlt,\Rmt,\Rrt,\Rlm,\Rmm,\Rrm,\Rlb,\Rmb,\Rrb}%
   \@print{front,\Flt,\Fmt,\Frt,\Flm,\Fmm,\Frm,\Flb,\Fmb,\Frb}%
   \@print{back,\Blt,\Bmt,\Brt,\Blm,\Bmm,\Brm,\Blb,\Bmb,\Brb}%
}
```

## 10.8 SaveRubikState command

\SaveRubikState The command `\SaveRubikState{`⟨*filename*⟩`}` saves the Rubik state to a named file in the format of a Rubik command (so it can then be processed by LaTeX). Note that in order to actually write a LaTeX command to a file without a trailing space one must prefix the command with the `\string` command (see Eijkhout (1992), p 238) [see refs Section 9].

Note that this macro uses the internal commands `\@comment` ('%%'), `\@commentone` ('%') and `\@print`. #1 is the output filename. We use several `\typeout` commands to write to the log file. An example of one of the lines of code we are trying to output to the `rubikstateNEW.dat` file is as follows:
`\RubikFaceUp{W}{W}{G}{W}{W}{G}{B}{B}{Y}%`

```
\newcommand{\SaveRubikState}[1]{%
\typeout{---NEW save command------------------}%
\typeout{---command = SaveRubikState{#1}}%
\typeout{---saving Rubik state data to file #1}%
\immediate\openout\outfile=#1%
\@print{\@comment filename: #1\@commentone}%
\@print{\string\RubikFaceUp%
   {\Ult}{\Umt}{\Urt}{\Ulm}{\Umm}{\Urm}{\Ulb}{\Umb}{\Urb}\@commentone}%
\@print{\string\RubikFaceDown%
   {\Dlt}{\Dmt}{\Drt}{\Dlm}{\Dmm}{\Drm}{\Dlb}{\Dmb}{\Drb}\@commentone}%
\@print{\string\RubikFaceLeft%
   {\Llt}{\Lmt}{\Lrt}{\Llm}{\Lmm}{\Lrm}{\Llb}{\Lmb}{\Lrb}\@commentone}%
\@print{\string\RubikFaceRight%
```

```
    {\Rlt}{\Rmt}{\Rrt}{\Rlm}{\Rmm}{\Rrm}{\Rlb}{\Rmb}{\Rrb}\@commentone}%
\@print{\string\RubikFaceFront%
    {\Flt}{\Fmt}{\Frt}{\Flm}{\Fmm}{\Frm}{\Flb}{\Fmb}{\Frb}\@commentone}%
\@print{\string\RubikFaceBack%
    {\Blt}{\Bmt}{\Brt}{\Blm}{\Bmm}{\Brm}{\Blb}{\Bmb}{\Brb}\@commentone}%
\immediate\closeout\outfile%
\typeout{----------------------------------------}%
}%
```

## 10.9 RubikRotation command

\RubikRotation

The `\RubikRotation[`⟨*integer*⟩`]{`⟨*comma separated sequence*⟩`}` command (a) writes the current Rubik state to the file `rubikstate.dat`, (b) writes the rotation sequence (either once or multiple times depending on the value of the optional integer argument), and then (c) CALLs the Perl script `rubikrotation.pl`. It also writes comments to the data file and also to the log file.

The way we allow the user to (optionally) process the main argument multiple times is simply by writing the associated output command multiple times to the output data-file. Consequently, we require the `\RubikRotation` command to allow a square-bracket optional argument (a non-negative integer) to specify the number of such repeats. In order to implement this optional argument facility we use two macros (countingloop and loopcounter) detailed by Feuersänger (2015) [see refs Section 9], as follows:

```
%% Two macros detailed by Feuersaenger (2015)
\long\def\@countingloop#1 in #2:#3#4{%
    #1=#2 %
    \@loopcounter{#1}{#3}{#4}%
}
%%--------------------------
\long\def\@loopcounter#1#2#3{%
    #3%
    \ifnum#1=#2 %
         \let\next=\relax%
    \else
         \advance#1 by1 %
         \def\next{\@loopcounter{#1}{#2}{#3}}%
    \fi
    \next
}
```

Having defined the above two macros we can now implement an optional argument (a repeat number) indicating the number of times we want the command to write the main argument to the output data file.

```
\newcommand{\RubikRotation}[2][1]{%
    \typeout{---TeX process----------------------------}%
    \typeout{---script = rubikrotation.sty v\RRfileversion\space (\RRfiledate)}%
    \typeout{---NEW rotation command}%
    \typeout{---command = RubikRotation[#1]{#2}}%
```

```
  \typeout{---writing current cube state to file rubikstate.dat}%
  \@openstatefile% open data file
  \@print{\@comment filename: rubikstate.dat}%
  \@print{\@comment written by rubikrotation.sty%
                          =v\RRfileversion\space (\RRfiledate)}%
  \@printrubikstate%
%% countingloop code from Feuersaenger (2015)
  \newcount\ourRRcounter%
  \@countingloop{\ourRRcounter} in 1:{#1}{%
        \immediate\write\outfile{rotation,#2}}%
  \@closestatefile% close data file
  \typeout{---CALLing Perl script (rubikrotation.pl)}%
  \immediate\write18{\rubikperlcmd}%
  \typeout{---inputting NEW datafile (data written by Perl script)}%
  \input{rubikstateNEW.dat}%
  \typeout{----------------------------------------}%
}
```

Note that the new `\ShellEscape` command implemented by the recent shellesc package is equivalent to `\immediate\write18` (see above), and so we probably ought to use `\ShellEscape` instead in future (and hence load shellesc automatically). At present, however, we leave the user to make sure that the shellesc package is actually available on their system.

## 10.10 ShowErrors command

`\ShowErrors` This command inputs the file `rubikstateERRORS.dat` (output by the Perl program).

```
\newcommand{\ShowErrors}{%
  \typeout{---ShowErrors: inputting file rubikstateERRORS.dat}%
  \VerbatimInput{rubikstateERRORS.dat}%
}
```

Since this command replaces the original command = `\ShowRubikErrors`, we will retain the original command for backward compatibility (for the moment at least).

```
\newcommand{\ShowRubikErrors}{\ShowErrors}
```

## 10.11 CheckState command

`\CheckState` This command triggers the Perl script to implement some simple error checking of the Rubik configuration (state). This command (a) writes the current Rubik state to the file `rubikstate.dat`, (b) writes the keyword 'checkstate' to the same file, and then (c) CALLs the Perl script. It also writes comments to the data file and also to the log file.

Note (1) that the command `\@printrubikstate` actually writes the current state to the file `rubikstate.dat`, (2) the keyword 'checkstate' triggers the Perl program to do a simple numerical check on the number of facelets with each colour.

Note also that the `\CheckState` command replaces the earlier `\CheckRubikState`, but we retain the old command for backwards compatibility (for the moment).

```
\newcommand{\CheckState}{%
   \typeout{---NEW check command------------------}%
   \typeout{---command = CheckState}%
   \typeout{---writing current cube state to file rubikstate.dat}%
   \@openstatefile% opens data file
   \@print{\@comment filename: rubikstate.dat}%
   \@printrubikstate%
         \immediate\write\outfile{checkstate}%
   \@closestatefile% close data file
   \typeout{---running Perl script (rubikrotation.pl)}%
   \immediate\write18{\rubikperlcmd}%
   \typeout{---inputting NEW datafile (data written by Perl script)}%
   \input{rubikstateNEW.dat}%
   \typeout{----------------------------------------}%
 }
\newcommand{\CheckRubikState}{\CheckState}
```

End of this package

⟨/rubikrotation⟩

# Index

Numbers written in italic refer to the page where the corresponding entry is described; numbers underlined refer to the code line of the definition; numbers in roman refer to the code lines where the entry is used.


**Symbols**

\% . . . . . 25
\@closestatefile . . . . . 49, 112, 132
\@comment 25, 39, 44, 45, 65, 104, 105, 129
\@commentone . . . . . 25, 65, 67, 69, 71, 73, 75, 77
\@countingloop . 82, 110
\@ifpackageloaded . . . . . 6, 9, 13, 17
\@loopcounter 84, 87, 93
\@openstatefile . . . . . 49, 103, 128
\@print 24, 39, 44, 45, 52, 53, 54, 55, 56, 57, 58, 65, 66, 68, 70, 72, 74, 76, 104, 105, 129
\@printrubikstate . . . . . 51, 107, 130

**A**

\advance . . . . . 92

**B**

\Blb . . . . . 58, 77
\Blm . . . . . 58, 77
\Blt . . . . . 58, 77
\Bmb . . . . . 58, 77
\Bmm . . . . . 58, 77
\Bmt . . . . . 58, 77
\Brb . . . . . 58, 77
\Brm . . . . . 58, 77
\Brt . . . . . 58, 77

**C**

\catcode . . . . . 25
\CheckRubikState . . 139
\CheckState . . . . *14*, 124
\closeout 40, 46, 50, 78

**D**

\def . . 2, 3, 25, 82, 87, 93
\Dlb . . . . . 54, 69
\Dlm . . . . . 54, 69
\Dlt . . . . . 54, 69
\Dmb . . . . . 54, 69
\Dmm . . . . . 54, 69
\Dmt . . . . . 54, 69
\Drb . . . . . 54, 69
\Drm . . . . . 54, 69
\Drt . . . . . 54, 69

**E**

\else . . . . . 91

**F**

\fi . . . . . 21, 94

\Flb . . . . . 57, 75
\Flm . . . . . 57, 75
\Flt . . . . . 57, 75
\Fmb . . . . . 57, 75
\Fmm . . . . . 57, 75
\Fmt . . . . . 57, 75
\Frb . . . . . 57, 75
\Frm . . . . . 57, 75
\Frt . . . . . 57, 75

**G**

\global . . . . . 25

**I**

\IfFileExists . . . . . 32
\ifluatex . . . . . 16
\ifnum . . . . . 89
\immediate . . . . 24, 38, 40, 43, 46, 49, 50, 64, 78, 111, 114, 131, 134
\input . . . . 33, 116, 136

**L**

\LaTeX . . . . . 19
\let . . . . . 90
\Llb . . . . . 55, 71
\Llm . . . . . 55, 71
\Llt . . . . . 55, 71
\Lmb . . . . . 55, 71
\Lmm . . . . . 55, 71
\Lmt . . . . . 55, 71
\long . . . . . 82, 87
\Lrb . . . . . 55, 71
\Lrm . . . . . 55, 71
\Lrt . . . . . 55, 71

**N**

\NeedsTeXFormat . . . . 4
\newcommand . . . . . . 22, 23, 24, 26, 27, 28, 29, 49, 50, 51, 60, 97, 119, 123, 124, 139
\newcount . . . . . 109
\newwrite . . . 37, 42, 48
\next . . . . . 90, 93, 95

**O**

\openout . 38, 43, 49, 64
\ourRRcounter . 109, 110
\outfile . . . . . 24, 37, 38, 40, 42, 43, 46, 48, 49, 50, 64, 78, 111, 131

**P**

\ProvidesPackage . . . 5

**R**

\relax . . . . . 90
\RequirePackage . . . . . 12, 15, 20
\Rlb . . . . . 56, 73
\Rlm . . . . . 56, 73
\Rlt . . . . . 56, 73
\Rmb . . . . . 56, 73
\Rmm . . . . . 56, 73
\Rmt . . . . . 56, 73
\Rrb . . . . . 56, 73
\RRfiledate 3, 5, 99, 106
\RRfileversion . . . . . 2, 5, 99, 106
\Rrm . . . . . 56, 73
\Rrt . . . . . 56, 73
\RubikFaceBack . . . . 76
\RubikFaceDown . . . . 68
\RubikFaceFront . . . 74
\RubikFaceLeft . . . . 70
\RubikFaceRight . . . 72
\RubikFaceUp . . . . . 66
\rubikpercentchar . . . . . 25, 26, 27
\rubikperlcmd . . . . . *6*, 29, 114, 134
\rubikperlname *6*, 28, 29
\RubikRotation . . . *8*, 81
\Rubikrotation . . . . 23
\rubikrotation . . . . 22

**S**

\SaveRubikState . *14*, 60
\SequenceInfo . . . . . *10*
\SequenceLong . . . . . *10*
\SequenceName . . . . . *10*
\SequenceShort . . . . *10*
\ShowErrors . . . . *15*, 119
\ShowRubikErrors . . 123
\space . 5, 26, 29, 99, 106
\string . . . . . 66, 68, 70, 72, 74, 76

**T**

\textsc . . . . . 22, 23
\typeout . . 7, 10, 14, 18, 31, 34, 36, 41, 47, 61, 62, 63, 79, 98, 99, 100, 101, 102, 113, 115, 117, 120, 125, 126, 127, 133, 135, 137

**U**

\Ulb . . . . . 53, 67
\Ulm . . . . . 53, 67
\Ult . . . . . 53, 67
\Umb . . . . . 53, 67
\Umm . . . . . 53, 67
\Umt . . . . . 53, 67
\Urb . . . . . 53, 67
\Urm . . . . . 53, 67
\Urt . . . . . 53, 67

**V**

\VerbatimInput . . . . 121

**W**

\write . . . . . 24, 111, 114, 131, 134