The RUBIKROTATION package

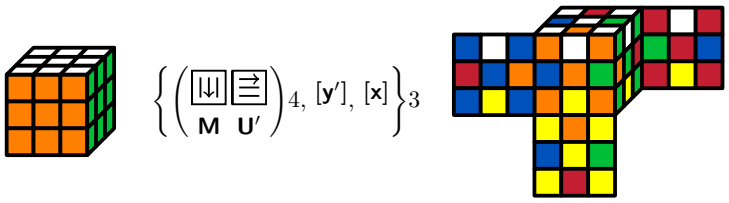
RWD Nickalls (dick@nickalls.org)
A Syropoulos (asyropoulos@yahoo.com)

This file describes version 3.0 (2015/09/25)
`www.ctan.org/pkg/rubik`

Abstract

The RUBIKROTATION package is a dynamic extension for the RUBIKCUBE package (both are part of the Rubik 'bundle'). The 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 a command for implementing some basic checking of the Rubik cube state (`\CheckRubikState`), and one for displaying any errors (`\ShowRubikErrors`).

The RUBIKROTATION package requires access to the TEX `write18` facility, which is enabled by using the `--shell-escape` command-line switch. The RUBIKROTATION package has been road-tested on a Microsoft platform (with MiKTeX and Strawberry Perl), on a Linux platform (Mandriva using TEXLive), and on a Solaris platform (OpenIndiana).



Contents

1 Introduction

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

The primary role of the RUBIKROTATION package is to implement a sequence of Rubik rotation moves on-the-fly using the `\RubikRotation` command. Consequently, this package requires the use of the `--shell-escape` switch to allow 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 (with MiKTeX and Strawberry Perl[1]), on a Linux platform (Mandriva using TEXLive), and on a Solaris platform (OpenIndiana).

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

```
\RubikRotation[]{}
\SaveRubikState
\CheckRubikState
\ShowRubikErrors
```

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        --the man file
Rubikrot-doc-figA.pdf
Rubikrot-doc-figB.pdf
Rubikrot-doc-figC.pdf
Rubikrot-doc-figD.pdf
examples.tex
examples.pdf
examples.sh
examples.bat
```

Note that the package includes an 'examples' file (`examples.tex`) as well as associated `.sh` (Linux) and `.bat` (Microsoft) batch files which can be used to facilitate processing the file. The main package documentation is the file `rubikrotation.pdf`.

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

```
pdflatex  rubikrotation.ins
```

[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.

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 your `/texmf-local/` directory tree. 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 into your current 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 4.3 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 this file (`rubikrotation.1`) would be typically located in the directory `/usr/share/man/man1`.

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.

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 3.0
      Usage: rubikrotation [-h] -i <input file> [-o <out file>]
      where,
      [-h]    gives this help listing
```

[2] Since the documentation includes a complicated indexing system as well a PDF 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.

```
[-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` and `rubikrotation.sty` in the TEX file preamble *after* loading the TikZ package (both Rubik packages require the TikZ package), for example, as follows:

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

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

4.1 Enabling the TEX 'shell' facility

In order to enable the TEX 'write18' facility (so it can run the Perl script) it is necessary to invoke (pdf)LATEX using the `--shell-escape` switch as follows:

```
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 &
```

4.2 Test file

An example tex file (which demonstrates the use of some of the package commands) is included in the package, namely:

```
example-rot.tex   (shows 8 worked examples)
```

This file needs to be run using `--shell-escape` switch; for example:

```
pdflatex  --shell-escape  example-rot.tex
```

Batch files (`.sh` for Linux, and `.bat` for Microsoft) are also provided to facilitate running the 'example' file. For example, on a Linux platform one would run a `bash` file as follows:

```
bash  example-rot.sh
```

If processing this file gives unexpected results (e.g., the cubes appear not to have experienced any rotations) check-out the associated log file to see if the operating system had any difficulties finding files etc.

4.3 Configuration-file

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 commands we may wish to adjust. The first is that defining the filename of the Perl script, namely `\rubikperlname`. The second is that defining the command-line call, namely `\rubikperlcmd`. The default definitions in `rubikrotation.sty` (which assume the Perl script is executable), 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 to use the `\space` command on the end of the backslash command (`\rubikperlname`) in order to force a following space—i.e., before the first command-line argument. 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 (called `rubikrotation.cfg`) which contains 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 in the configuration-file, as follows:

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

PLACING THE CONFIGURATION-FILE: The simplest arrangement is just to include the `.cfg` file in the working 'test' 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.

However, using commands which influence the Rubik colour state (e.g., the \RubikRotation command) 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 accessable 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 \ShowRubikErrors command (see the notes on this command below).

5.1 \RubikRotation command

\RubikRotation The \RubikRotation[⟨*integer*⟩]{⟨*comma separated sequence*⟩} command processes a comma separated sequence of rotations, and returns the final state. The optional [⟨*integer*⟩] argument specifies the number of times the command should be repeated.

For example, if we wanted to see the effect of the rotations **R, R, L, U, D** on a solved Rubik cube, we could use the following commands.

```
\RubikCubeSolved   % sets up the colours for a solved cube state
\RubikRotation{R2,L,U,D}
\begin{tikzpicture}[scale=0.7]
   \DrawRubikCubeRU
\end{tikzpicture}%
```

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 any error messages) to the file rubikstateNEW.dat, which is

then input on-the-fly by the LaTeX file. This new Rubik state can then either used as the input for another `\RubikRotation`, or used to generate a graphic image of the cube.

A given rotation sequence can be repeated multiple times, say n times, by using the optional `[n]`. For example, the following two commands are equivalent:

```
\RubikRotation[3]{x,R,U}
\RubikRotation{x,R,U,x,R,U,x,R,U}
```

5.1.1 Sequences as macros

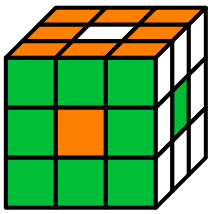
Macros which are arguments of the TeX `\write` command are expanded on writing (Eijkhout 1992, § 30.2.3, p. 238)[see refs Section 8]. Consequently we are able to use a sequence-defining macro as an argument for the `\RubikRotation` command. In fact this is very convenient, since it allows one to store lots of different rotation sequences by name alone.

For example, we can 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 refs Section 8], as follows:

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

With this new `\sixspot` command 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):

```
\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}`.

5.1.2 Arguments prefixed with *, [,]

If any of the comma separated arguments is prefixed with either *, [or] they are interpreted as an inactive 'string', and not as a rotation. This feature therefore allows a string argument to be used as a label, which can be very useful.

For example, we can use this facility to label 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 feature in the `\sixspot` command, as follows,

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

since this has the great advantage of making the label-name visible in the log file. For example, the following command, which uses the rotations **x**, **x** and **y** to initially rotate the 'solved' cube before applying the 'sixspot' sequence of rotations,

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

will then be represented in the log file as

```
...command=rotation,x2,y,[sixspot],U,Dp,R,Lp,F,Bp,U,Dp
...arguments passed to 'rotation' sub = x2 y [sixspot] U Dp R Lp F ...
...rotation x OK (= rrR + rrSr + rrLp)
...rotation x OK (= rrR + rrSr + rrLp)
...rotation y OK (= rrU + rrSu + rrDp)
...[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
```

In this way, several named rotation sequences can 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.

To this end, the `ForEachX` macro—from the `forarray` package—can be very useful. For example, this macro is central to the following example macro for typesetting a rotation sequence using the Rubik `\rr` command:

```
\def\x{\thislevelitem}
\def\xcount{\thislevelcount}

\newcommand{\showseq}[1]{%
  \ForEachX{,}{%
    \ifthenelse{\xcount=1}{\texttt{\x}}{,\ \rr{\x}}%
    }{#1}.
}
```

Now, for a sequence defined as `\newcommand{\myseq}{[myseq],U,D,Lwp,R}`, then the command `\showseq{\myseq}` will result in the following output:

`[myseq]`, **U**, **D**, **Lwp**, **R**.

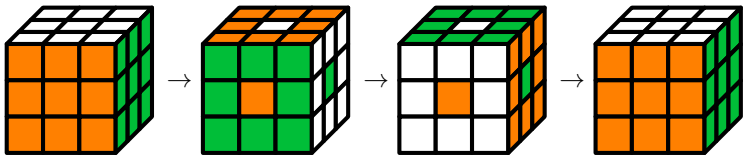
Note that we are able to typeset the name `[myseq]` differently from the remaining sequence since the counter `\xcount` allows us to locate the first item. However,

this macro as it stands needs further development in order to handle sequence elements with a terminal digit (e.g., `R2`)—the macro will need to first expand this to `R, R` if the Rubik commands are to be used.

5.1.3 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 (`\sixspot`)$3 \equiv 0$, and can be nicely illustrated by implementing the following pseudocode:

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



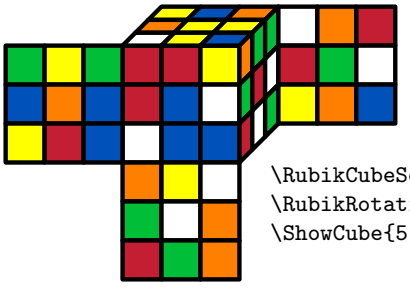
5.1.4 Random rotations

The `\RubikRotation` command can also be used to scramble the cube using a random sequence of rotations. If the first argument is the lowercase 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; otherwise a default value of 50 is used (for example, if the second argument is not an integer). 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 (see line 583) we currently have

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

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



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

Note that in this particular example (above), only the `\Draw..` command is inside the TikZ picture environment (i.e., inside the `\ShowCube` command). Note 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{*sixspot,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 later, when required, simply by inputting the file as follows:

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

5.3 `\CheckRubikState` command

`\CheckRubikState` Since it is easy to inadvertently define an invalid Rubik cube (e.g., enter an invalid number of, say, yellow facelets), this command 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 6. The results are written to the the `.log` file, and displayed under the graphic if the `\ShowRubikErrors` command is used.

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

```
\CheckRubikState%
```

Note that such a check is implemented automatically with each `\RubikRotation` command.

5.4 `\ShowRubikErrors` command

`\ShowRubikErrors` Any errors which arise can be made visible using the command `\ShowRubikErrors`. 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—all this detail can also be found in the `.log` file.

Consequently, this 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 (if there are several such environments), 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. Run the test file `example-rot.tex` to see an example of the use of this command.

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 `\ShowRubikErrors` is used after the TikZ picture environment.

7 General overview

When LaTeX processes `rubikrotation.sty` the following steps are implemented.

1. A check is made to see if `fancyvrb.sty` is loaded: if not then this package is loaded if it is available (this package is required for inputting the file `rubikstateERRORS.dat`).

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' 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 it first writes the current colour configuration of each face (the 'rubik state') to the temporary file `rubikstate.dat` (to 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 would be as follows:

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

A `\CheckRubikState` command triggers the same sequence of events except 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 to Perl, the script `rubikrotation.pl` starts by loading the current rubikstate (prompted by the keywords `up`, `down`, `left`, `right`, `front`, `back`, in the file `rubikstate.dat`). Next the Perl script performs some basic checks (prompted by the key word `checkstate`), and then it 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.

Control then reverts to LaTeX which then inputs the file `rubikstateNEW.dat`. If there are more `\RubikRotation` commands then this cycle repeats accordingly. Eventually a `\Draw...` command of some form is reached and the final rubikstate is drawn in a TikZ picture environment.

If the TikZ picture environment is followed by a `\ShowRubikErrors` command, then a 'verbatim' copy of the `rubikstateERRORS.dat` file is displayed immediately under the graphic. Once the graphic is error-free, then the `\ShowRubikErrors` 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 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 5.1.1]

- 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]

- Nickalls RWD and Syropoulos A (2015). The RUBIKCUBE package.
  http://www.ctan.org/pkg/rubik,

- Reid M. Patterns. http://www.cflmath.com/Rubik/patterns.html [re: sequences as macros; in Section 5.1.1]

9 Change history

- 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 `\RubkRotation` command is prefixed with either a * or [or] character then it is not actioned as a rotation (see Section 5.1.2).

  — 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 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.

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

```
1 ⟨*rubikrotation⟩
2 \def\RRfileversion{3.0}%
3 \def\RRfiledate{2015/09/25}%
4 \NeedsTeXFormat{LaTeX2e}
5 \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 10.10).

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

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 8], 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 4.3).

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

`\rubikperlcmd` This holds the command-line code for calling the Perl script (assumed to be an executable file). 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 this .sty file actually specifies 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 from the file `rubikstateNEW.dat`.

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

Remember to use the `\space` command following the LaTeX backslash command in order to generate the mandatory space between it and 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 4.3).

10.3 Configuration file

If a configuration file exists (`rubikrotation.cfg`) then input it here, i.e., *after* defining the `\rubikperlname` and `\rubikperlcmd` commands 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 `\ShowRubikErrors`.

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 rubikstateERRORS.dat}%
\@print{\@comment ---(RR.sty v\RRfileversion): comments output by Perl script}%
\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 append the key-word checkstate to the end of the file in order to trigger the Perl script to implement its checkstate subroutine.

The actual state is simply an ordered sequence of the faces and the colours associated with each facelet of that 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{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}%
  \@print{checkstate}%
```

```
}
```

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 8].

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 the line 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.

In order to allow the user to (optionally) write the main argument multiple times to the output data-file, we require this command to allow an optional argument (a non-negative integer) to specify the number of such repeats. In order

to implement an optional argument facility we use two macros (countingloop and loopcounter) detailed by Feuersänger (2015) [see refs Section 8], 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{---NEW rotation command-------------------}%
   \typeout{---command = RubikRotation{#1}}%
   \typeout{---writing current Rubik 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{---running Perl script (rubikrotation)}%
   \immediate\write18{\rubikperlcmd}%
   \typeout{---inputting NEW datafile (written by Perl script)}%
   \input{rubikstateNEW.dat}%
   \typeout{----------------------------------------}%
}
```

10.10 ShowRubikErrors command

\ShowRubikErrors This command inputs the file `rubikstateERRORS.dat`.

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

```
}
```

10.11 CheckRubikState command

\CheckRubikState

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`, and then (b) CALLs the Perl script. It also writes comments to the data file and also to the log file..

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

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.