Documentation of `rubikrotation.pl` version 5.0

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

Last revised: 25 February 2018
This file: `rubikrotationPL.pdf`*

Contents

1 Overview **4**
1.1 Input datafile (`rubikstate.dat`) . . . 4
1.2 Program actions . . . 6
1.3 Processing a RubikRotation sequence . . . 6
1.4 `RubikRotation` command . . . 7
1.5 References . . . 7

2 Program process example **7**

3 The code **14**
3.1 MAIN . . . 18
3.2 rotation . . . 32
3.3 random . . . 63
3.4 writestate . . . 66
3.5 ErrorMessage . . . 71
3.6 gprint . . . 71
3.7 checkstate . . . 72
3.8 rr-overview . . . 74
3.9 rrR . . . 75
3.10 rrSr . . . 78
3.11 rrLp . . . 80
3.12 rrU . . . 82
3.13 rrSu . . . 85
3.14 rrDp . . . 86
3.15 rrF . . . 89
3.16 rrSf . . . 91
3.17 rrBp . . . 92
3.18 rr-derivative subs . . . 95
3.19 rrRp . . . 95
3.20 rrRw . . . 95
3.21 rrRwp . . . 95
3.22 rrRs . . . 96
3.23 rrRsp . . . 96
3.24 rrRa . . . 96
3.25 rrRap . . . 96
3.26 rrL . . . 96
3.27 rrLw . . . 96

*Part of the Rubik bundle v5.0: available from `http://www.ctan.org/pkg/rubik`

3.28 rrLwp . . . 96
3.29 rrLs . . . 96
3.30 rrLsp . . . 96
3.31 rrLa . . . 97
3.32 rrLap . . . 97
3.33 rrUp . . . 97
3.34 rrUw . . . 97
3.35 rrUwp . . . 97
3.36 rrUs . . . 97
3.37 rrUsp . . . 97
3.38 rrUa . . . 97
3.39 rrUap . . . 98
3.40 rrD . . . 98
3.41 rrDw . . . 98
3.42 rrDwp . . . 98
3.43 rrDs . . . 98
3.44 rrDsp . . . 98
3.45 rrDa . . . 98
3.46 rrDap . . . 98
3.47 rrFw . . . 99
3.48 rrFp . . . 99
3.49 rrFwp . . . 99
3.50 rrFs . . . 99
3.51 rrFsp . . . 99
3.52 rrFa . . . 99
3.53 rrFap . . . 99
3.54 rrB . . . 99
3.55 rrBw . . . 99
3.56 rrBwp . . . 100
3.57 rrBs . . . 100
3.58 rrBsp . . . 100
3.59 rrBa . . . 100
3.60 rrBap . . . 100
3.61 rrSup . . . 100
3.62 rrSd . . . 100
3.63 rrSdp . . . 100
3.64 rrSl . . . 100
3.65 rrSlp . . . 101
3.66 rrSrp . . . 101
3.67 rrSfp . . . 101
3.68 rrSb . . . 101
3.69 rrSbp . . . 101
3.70 rubikmod . . . 101
3.71 cleanstring . . . 103
3.72 cutinfoblock . . . 104
3.73 fixrepeatelement . . . 106
3.74 repeat . . . 108
3.75 quitprogram . . . 111
3.76 showarray . . . 111
3.77 cleanarray . . . 111
3.78 restorebrackets . . . 112

3.79 infoblockcolon 112
3.80 RemoveAllSpaces 115
3.81 CheckSyntax 116
3.82 inverse 123

1 Overview

This Perl program (`rubikrotation.pl`), is part of the LaTeX rubikrotation package, which itself is part of the Rubik bundle (available from `http://www.ctan.org/pkg/rubik`). A general overview of the interaction between `rubikrotation.sty` and this Perl program is presented in the RUBIKROTATION documentation (see Section *General overview*).

This Perl program reads (as input) a formatted 'data' file (typically the file `rubikstate.dat` —see below). This data-file defines the current state of the Rubik cube; each line beginning with a specified 'keyword'. After processing the input data-file, the program writes the final state to a text-file (typically the file `rubikstateNEW.dat` when used with the rubikcube package), and writes any error messages to the file `rubikstateERRORS.dat`.

Usage: Version 3 onwards uses the input and output filenames specified by the CALLing command-line, the usage for the executable form being as follows:

```
$ rubikrotation  -i <input filename>  [-o <output filename>]
```

If no output filename is specified, then the default filename `rubikOUT.txt` is used. When used in conjunction with the rubikcube package, then `rubikcube.sty` calls the program and sets the input filename as `rubikstate.dat` and the output filename as `rubikstateNEW.dat`.

Functionality: The program processes the argument of the LaTeX `\RubikRotation{}` command, and returns a file `\rubikstateNEW.dat` containing the final rubik state as well as other supporting data. For example, a typical `\RubikRotation{}` command can be made up of various bracket delimited sections, as well as a sequence of comma separated Rubik rotation codes, as follows:

```
\RubikRotation[3]{[test1],L,R,U,D,([seqA],U,D2,F3)3,<inverse>,<(pf*) //(C2){h2}>}
```

Version 4 onwards allows additional functionality with regard to the RubikRotation sequence of rotation-codes, namely: (a) repeating elements of the form: `..,([repeatname],U,D2,F3)n,..` where n is an integer, (b) 'info-blocks' for carrying special information regarding the sequence (meta data), and delimited by a balanced pair of angle brackets; for example, `..,<(pf*) //(C2){h2}>` (see the RUBIKROTATION documentation for details). All delimiters in the rotation sequence must be balanced, otherwise the program will terminate. If the keyword 'inverse' appears inside an info-block, then this triggers the program to generate the (mathematically) inverse of the sequence of rotation-codes (see the RUBIKROTATION documentation for details).

1.1 Input datafile (`rubikstate.dat`)

This Perl program reads as input the plain-text file `rubikstate.dat` which is output in response to either a `\RubikRotation` command (see `rubikrotation.sty`) or a `\TwoRotation` command (see `rubiktwocube.sty`) in the `.tex` file. The following `rubikstate.dat` file was generated in response to the command: `\RubikRotation{[mytestseq],L,R2,(U,D)3,F,B,<(pf*)>}`

```
%% 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,[mytestseq],L,R2,(U,D)3,F,B,<(pf*)>
```

Each line of the input datafile `rubikstate.dat` is a comma-separated sequence of elements or codes (spaces are tolerated next to commas); the first element of each line is a 'keyword' which informs the Perl program regarding how to process the subsequent arguments. This data-file defines the current (initial) state of the Rubik cube (using the 'face' keywords up, down, left, right, front, back), and may include a range of other keywords (cubesize, checkstate, rotation, random) which are used to trigger various subroutines to process the input data accordingly.

cubesize

The keyword `cubesize` is followed by either the word `two` (indicating we are dealing with a TWOcube = mini cube) or by the word `three` (indicating we are dealing with a THREEcube = Rubik cube). The word `three` is written by the `\RubikRotation` command (see `rubikrotation.sty`), and the word `two` is written by the `\TwoRotation` command (see `rubiktwocube.sty`). This (second) word is allocated to the string variable `$cubesize`, and hence the Perl programme is able to use this variable to distinguish between the two cubes.

up, down, left, right, front, back

If the keyword is either `up, down, left, right, front, back` then the subsequent colour codes are written to the associated face cubie variables.

checkstate

If the keyword is `checkstate` then a simple error check of the state of the Rubik cube colour configuration is undertaken.

rotation

The keyword 'rotation' is associated with two different line formats which are distinguished according to whether or not the second element is the word 'random'. If the second element is the word 'random' then the command is interpreted as an instruction to implement a sequence of n random rotations with a view to scrambling the cube; for example, the following line is an instruction to generate 120 random rotations.

```
rotation,random,120
```

Note that the array of valid rotations from which such random rotations can be chosen depends on the particular cube (as indicated via the `cubesize` parameter in the `rubikstate.dat` file). This is because a THREEcube (Rubik cube) allows both middle and double-slice rotations, neither of which are valid for a TWOcube (see the 'random' subroutine for details of the arrays used).

If the second element is *not* 'random' then we are dealing with a rotation sequence. For example, the following line is an instruction to generate the sequence of rotations `L,R2,(U,D)3,F,B`:

```
rotation,[mytestseq],L,R2,(U,D)3,F,B,<(pf*)>
```

Note that brackets of various types are allowed; the rules are as follows. All brackets on a line must be balanced. Balanced pairs of brackets can be positioned anywhere. An unbalanced bracket will generate an early fatal error (as syntax checking is done at an early stage) causing the program to terminate with an appropriate error message.

The contents of square brackets are not processed as rotations, and hence can be used to indicate a name (only the first such bracket will be allocated as a 'name'), or tag. Square brackets must not contain commas (as commas are reserved for separating sequence elements). In principle users should be able to include any characters inside 'nameblocks' (bounded by square brackets [...]) and 'infoblocks' (bounded by angle brackets <...>).

Angle brackets are used to carry sequence information or meta-data (this structure is known as an 'info-block'). Although info-blocks can be placed anywhere, the macros in the 'rubikpatterns' database typically have info-blocks positioned as the final argument of a rotation sequence. There are no content restrictions for balanced angle brackets (= info-blocks)

Curved brackets indicate a repeated rotation sequence (a 'repeat block'); a trailing integer n ($n \geq 0$) is the repeat number, and if absent it is actioned as $n = 1$.

1.2 Program actions

The Perl program is called by a system command issued by the `rubikrotation.sty` and triggered by either the `\RubikRotation` command or the `\TwoRotation` command in the `.tex` file. The Perl program proceeds by reading the file `rubikstate.dat` line-by-line, and acting according to the keyword (first element of the line). Each input line is checked for syntax at an early stage. Significant syntax errors (unbalanced or nested brackets, missing commas) result in the program issuing appropriate error messages and then terminating cleanly.

The final rubik state, together with a lot of general information and error-messages are all written to the file `rubikstateNEW.dat`. All lines containing the word 'ERROR' are also written to the file `rubikstateERRORS.dat`. Since the file `rubikstateNEW.dat` is automatically input by the `rubikrotation.sty` as the `\RubikRotation` command terminates, then all this information also appears in the LaTeX log file.

1.3 Processing a RubikRotation sequence

We will now focus on the key actions associated with processing a typical rotation sequence.

1: The sequence is checked for syntax errors at an early stage, to reduce unnecessary processing.

2: A copy of the original sequence → `$SequenceShort`

3: Infoblock(s) are removed and concaternated → `$SequenceInfo`

4: Check for any 'repeat blocks'. If detected, then any associated `( , )` are converted to `{ ; }` respectively. This then allows any repeat-blocks to be safely processed within a string of comma-separated rotation elements which can be fed into the 'rotation' subroutine, where they will be expanded accordingly. For example `,(L,R)2,` → `,{L;R}2,` → `,L,R,L,R,` etc.

5: The sequence of comma separated elements (some of which may be 'reformulated' repeat-blocks—see above) is now processed into an array, the elements of which are then passed to the 'rotation' subroutine for further processing. Any repeat-blocks are then detected and their elements sent back into the 'rotation' subroutine the appropriate number of times.

6: Square-bracket elements (nameblocks) are detected: the first such → `$SequenceName`.

7: All rotation codes are processed mod (4), and each instance is then appended to a so-called 'Long' string → `$SequenceLong`.

8: The four `$Sequence...` variables (see above) are written to the file `rubikstateNEW.dat` as the macros `\SequenceInfo`, `\SequenceName`, `\SequenceShort`, `\SequenceLong`. Note that this makes them available to the LaTeX document, since the `\RubikRotation` command (in the `rubikrotation.sty`) automatically inputs the file `rubikstateNEW.dat` as control passes back to LaTeX when the Perl program terminates. See also the code for the `\RubikRotation` command below.

9: All output data (eg rubik state, error messages, sequence macros) is written to the file `rubikstateNEW.dat`. This file is input by LaTeX (see above) and hence all error messages also appear in the log file.

10: Error messages are collected in an array and written to the file `rubikstateNEW.dat` and also to the file `rubikstateERRORS.dat`. The LaTeX command `\ShowErrors` results in a verbatim copy of the `rubikstateERRORS.dat` file being input into the document.

1.4 RubikRotation command

The following \RubikRotation command (from the file rubikrotation.sty), shows the code for writing the file rubikstate.dat, calling the Perl program, and finally, the inputting of the file rubikstateNEW.dat.

```
 \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{----------------------------------------}%
 }
```

1.5 References

The following texts are cited in the Perl programme.

- Holzner S (1999). Perl core language little black book. (The Coriolis Group, LLC, Scottisdale, Arizona, USA) (www.coriolis.com); pp 492.
- Wall L, Christiansen T and Orwant J (2000). Programming Perl. 3rd Ed; (O'Reilly & Associates Inc.)
- Christiansen T and Torkington N (1999). Perl cookbook. 1st Ed + corrections; (O'Reilly & Associates Inc.)
- chromatic with Conway D and Poe C (2006). Perl hacks. (O'Reilly Media Inc., Sebastopol, CA 95472, USA).

2 Program process example

```
\RubikCubeSolvedWY
\ShowCube{1.6cm}{0.4}{\DrawRubikCubeRU}%
    \RubikRotation{[test],L,(R,F)2,D}
\ShowCube{1.6cm}{0.4}{\DrawRubikCubeRU}%
```

We can see how the Perl programme processes the above `\RubikRotation` command by capturing the screen output; in the following example we can see how it starts by processing the initial Rubik state information (relating to the colour state of each face of the WY solved cube) and then expands the repeat block `(R,F)2`. The lines with a leading ... are also copied to the TEX log file.

```
...PERL process.................................
...script = rubikrotation.pl v5.0 (25 February 2018)
...reading the current cube state (from File: rubikstate.dat)

 TOP --------- (new line)

 SUB CheckSyntax
 dataline = cubesize,three
 left and right <> = 0, 0
 left and right [] = 0, 0
 left and right () = 0, 0
 leftsum, rightsum = 0, 0
 lenstring = 14
 syntax OK; brackets balanced OK
 done

 dataline array = cubesize three
...
...command = cubesize,three
...cube = THREEcube
...

 TOP --------- (new line)

 SUB CheckSyntax
 dataline = up,W,W,W,W,W,W,W,W,W
 left and right <> = 0, 0
 left and right [] = 0, 0
 left and right () = 0, 0
 leftsum, rightsum = 0, 0
 lenstring = 20
 syntax OK; brackets balanced OK
 done

 dataline array = up W W W W W W W W W
......up,W,W,W,W,W,W,W,W,W

 TOP --------- (new line)

 SUB CheckSyntax
 dataline = down,Y,Y,Y,Y,Y,Y,Y,Y,Y
 left and right <> = 0, 0
 left and right [] = 0, 0
 left and right () = 0, 0
 leftsum, rightsum = 0, 0
 lenstring = 22
```

```
 syntax OK; brackets balanced OK
 done

 dataline array = down Y Y Y Y Y Y Y Y Y
....down,Y,Y,Y,Y,Y,Y,Y,Y,Y

 TOP --------- (new line)

 SUB CheckSyntax
 dataline = left,B,B,B,B,B,B,B,B,B
 left and right <> = 0, 0
 left and right [] = 0, 0
 left and right () = 0, 0
 leftsum, rightsum = 0, 0
 lenstring = 22
 syntax OK; brackets balanced OK
 done

 dataline array = left B B B B B B B B B
....left,B,B,B,B,B,B,B,B,B

 TOP --------- (new line)

 SUB CheckSyntax
 dataline = right,G,G,G,G,G,G,G,G,G
 left and right <> = 0, 0
 left and right [] = 0, 0
 left and right () = 0, 0
 leftsum, rightsum = 0, 0
 lenstring = 23
 syntax OK; brackets balanced OK
 done

 dataline array = right G G G G G G G G G
...right,G,G,G,G,G,G,G,G,G

 TOP --------- (new line)

 SUB CheckSyntax
 dataline = front,O,O,O,O,O,O,O,O,O
 left and right <> = 0, 0
 left and right [] = 0, 0
 left and right () = 0, 0
 leftsum, rightsum = 0, 0
 lenstring = 23
 syntax OK; brackets balanced OK
 done

 dataline array = front O O O O O O O O O
...front,O,O,O,O,O,O,O,O,O
```

```
 TOP --------- (new line)

 SUB CheckSyntax
 dataline = back,R,R,R,R,R,R,R,R,R
 left and right <> = 0, 0
 left and right [] = 0, 0
 left and right () = 0, 0
 leftsum, rightsum = 0, 0
 lenstring = 22
 syntax OK; brackets balanced OK
 done

 dataline array = back R R R R R R R R R
....back,R,R,R,R,R,R,R,R,R

 TOP --------- (new line)

 SUB CheckSyntax
 dataline = rotation,[test],L,(R,F)2,D
 left and right <> = 0, 0
 left and right [] = 1, 1
 left and right () = 1, 1
 leftsum, rightsum = 2, 2
 lenstring = 26
 syntax OK; brackets balanced OK
 done

 dataline array = rotation [test]  L  (R F)2  D
...
...rotation keyword
 SUB checkstate
...checking state of cube
...cubiesum = 54 (Red=9, Or=9, Ye=9, Gr=9, Bl=9, Wh=9, X=0)
 done

...
...command = rotation,[test], L, (R,F)2, D
...SUB InfoblockColon
 no <infoblock> found.

...dataline = rotation,[test], L, (R,F)2, D
 SUB fixrepeatelement
 reformatting any repeat elements...
 p = 20, q = 24
 first repeat string = (R,F)
 length of repeat string = 5
...new repeat string = {R;F}
 frontstring = rotation,[test], L,
 backstring = 2, D
 new dataline = rotation,[test], L, {R;F}2, D
 done
```

```
 repairing braces and semicolon--> ()
 SequenceShort = [test], L, (R,F)2, D
 processing rotation arguments: = [test]  L  {R;F}2  D (n= 4)
 CALLing SUB rotation
 SUB rotation
...[test] is a label OK
 SUB rotation
 SUB rubikmod
...rotation L,   OK (= Lp3)
 SUB rotation
 repeat block found        = {R;F}2
 repeat block reformulated = {R,F}2
...Expanding: (R,F)2 ...
 CALLing SUB: repeat({R,F}2)
 SUB repeat
 repeatcode = R,F
 lenrepeatstring = 6
 lenrepeatcode = 3
 p = 0
 q = 4
 repeatnumber = 2
 insert = R,F,R,F
 done

 expanded_repeatcode = R,F,R,F

 CALLing SUB rotation
 sending repeat element R to rotation SUB
 SUB rotation
 SUB rubikmod
...rotation R,   OK
 sending repeat element F to rotation SUB
 SUB rotation
 SUB rubikmod
...rotation F,   OK
 sending repeat element R to rotation SUB
 SUB rotation
 SUB rubikmod
...rotation R,   OK
 sending repeat element F to rotation SUB
 SUB rotation
 SUB rubikmod
...rotation F,   OK
 SUB rotation
 SUB rubikmod
...rotation D,   OK (= Dp3)
 closing down:  writing state...
 SUB writestate
 Perl output file written OK
<<omx.cmap>>
```

```
LaTeX Warning: No \author given.

---TeX process-------------------------
---script = rubikrotation.sty v5.0 (2018/02/06)
---NEW rotation command
---command = RubikRotation[1]{[test], L, (R,F)2, D}
---writing current cube state to file rubikstate.dat
---CALLing Perl script (rubikrotation.pl)
---inputting NEW datafile (data written by Perl script)
(./rubikstateNEW.dat

...PERL process.............................
...script = rubikrotation.pl v5.0 (25 February 2018)
...reading the current cube state (from File: rubikstate.dat)
...
...command = cubesize,three
...cube = THREEcube
...
......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
...
...rotation keyword
...checking state of cube
...cubiesum = 54 (Red=9, Or=9, Ye=9, Gr=9, Bl=9, Wh=9, X=0)
...
...command = rotation,[test], L, (R,F)2, D
...dataline = rotation,[test], L, (R,F)2, D
...[test] is a label OK
...rotation L, OK (= Lp3)
...Expanding: (R,F)2 ...
...rotation R, OK
...rotation F, OK
...rotation R, OK
...rotation F, OK
...rotation D, OK (= Dp3)
...writing new cube state to file rubikstateNEW.dat
...SequenceName = test
...SequenceInfo =
...SequenceShort = [test], L, (R,F)2, D
...SequenceLong = L,R,F,R,F,D
)
----------------------------------------
[1{/usr/local/texlive/2017/texmf-var/fonts/map/pdftex/updmap/pdftex.map}]
(./test-sidebars.aux) ){/usr/local/texlive/2017/texmf-dist/fonts/enc/dvips/lm/l
m-ec.enc}</usr/local/texlive/2017/texmf-dist/fonts/type1/public/lm/lmr10.pfb></
usr/local/texlive/2017/texmf-dist/fonts/type1/public/lm/lmr12.pfb></usr/local/t
```

```
exlive/2017/texmf-dist/fonts/type1/public/lm/lmr17.pfb>
Output written on test-sidebars.pdf (1 page, 69493 bytes).
Transcript written on test-sidebars.log.
```

3 The code

```
#!/usr/bin/env perl
##
use Carp;
use Fatal;
use warnings;
##
our $version = "v5.0   (25 February 2018)";
##
##  rubikrotation.pl
##  VERSION 5.0
##
##  Copyright February 2018,
##  RWD Nickalls  (dick@nickalls.org)
##  A Syropoulos  (asyropoulos@yahoo.com)
##
##  HISTORY
##------------------------------
##  v5.0 (25 February 2018)
##------------------------------
##
##  4.2h  (07 February 2018)
##       --- adjusted the no of leading dots for the gprint command
##           when writing the up, down,.. colour state to the log file
##           so as to make the array form a square block (ie easier to read)
##           (in MAIN)
##
##  4.2g
##      --- 29 October 2017  added syntax checking for (( )) /inside/ squarebrackets
##                  (in SUB checksyntax)
##      --- 24 October 2017  changed {};--> (), in SequenceNameNew (in SUB writestate)
##                  this repairs these chars back to their original state.
##      --- 22 October 2017  minor adjustments to syntax checking (in SUB checksyntax)
##               to allow some extra chars in the [name] and <info> blocks.
##               Ideally, we want to be able to use /any/ chars inside these infoblocks.
```

```
##
##  4.2f
##      --- 05 Oct 2017 bugfix: added a ShowSequence [\space] bug fix in SUB checksyntax
##      --- 04 Oct 2017 adjusted brackets < > error messages (lines 2643--2659)
##
##  4.2e: (29 Sept 2017)
##      --- added a ‘‘Western’’ notation filter (provisional \& works)
##
##  4.2d: (10 August 2017)
##      --- placed  a checkstate() command inside the rotation keyword
##          and stopped TEX writing the keyword checkstate to the rubikstate.dat file
##
##  4.2c: (2 August 2017)
##      --- added new rubikkeyword "cubesize" to hold cube size (three or two)
##          so we can tell which sort of cube  is being processed
##          We can use this to detect when using the TwoRotation command
##          (for the TWOcube) vs when using the RubikRotation command
##          (for the THREEcube); for eample with regard to random rotations
##          (see random SUB; see RubikTwoCube.sty)
##
##  4.2b: (1 Aug 2017)
##      --- Removed the random,0 option --> n=50 (random SUB)
##          a zero or missing integer now generates an error message.
##
##  4.2a: (28 July 2017)
##      --- bugfix: error if spaces in RubikRotation{random,n} string from LaTeX.
##          Fixed to accommodate spaces, and uppercase random (lines 366 -- 388 approx)
##
##------------------------------
##  v4.0 (3 March 2017)
##------------------------------
##  changes in v3.6 (January 2017)
##   --- included Jaap Rm and Rc notation
##   --- new sub for improved expansion of mod-4 multiples of rotations (Oct 2016)
##   --- restructured to facilitate processing arrays through the rotation sub
##   --- included option for an <info> block
```

```
##   --- included Randelshofer superset ENG 3x3 notation
##   --- implemented an 'inverse' mode
##   --- improved syntax checking
##   --- used perltidy to polish the program layout
##       (but only when making  the pdf documentation-- see file rubikrotationPL.pdf)
##   --- included a lot of new subroutines
##--------------------------------
##  changes in v3.2:
##  v3.2h: (2 Oct 2016)
##          improved the mod 4 routine using  SUB rubikmod()
##          improved comments to log file re: rotation processing
##
##  v3.2e:(25 Sept 2016)
##           changed some command names: use short & long for the Rubik R2 --> R,R code
##           (more intuitive than Clean)
##           \Sequence{} -->      SecquenceShort{}
##           \SequenceClean{} --> SecquenceLong{}
##           removed the [ and ] around [name] variable
##
##  v3.2d: changed the returned command names (removed the Rotation part to keep it simple)
##           \Sequence{}  = orig seq  + NO NAME
##           \SequenceName{} = NAME only
##           \SequenceClean{} = clean seq + NO NAME
##
##  v3.2c: added new commands:
##           \RotationSequenceName{}
##           \RotationSequenceClean{}
##
##  v3.2a: added a \RubikSeqNEW{...}  output line in the output file
##        to facilitate typesetting the rotation sequence (works OK just now)
##
##  v3.2:  --- added leading ... to the comments written by the <writestate> sub
##        (the ... code indicates that comments are written by the Perl script)
##        --- changed the word program, prog --> script
##--------------------------------
##  changes in v3.0:
```

```
##   --- accepts command-line arguments for input (mandatory) and output (optional) filenames
##       default output filename is: rubikOUT.txt
##   --- included the symbols [ and ] to denote a rotation-name label (ie as well as *)
##   --- fixed some of the variable definitions (as highlighted by <use strict> pragma)
##-------------------------------
##  changes in v2.3:
##   --- accepts a single  commandline argument (datafilename)
##   --- uses the standard modules Carp and Fatal (give extra  line info on error)
##-------------------------------
##  changes in v2.2:
##   --- changed licence --> LatexPP
##   --- included random n errors in ERROR messages (lines 492--495)
##   --- included version number in error message
##-------------------------------
#
# This file is part of the LaTeX  rubikrotation package, and
# requires rubikcube.sty and rubikrotation.sty
#
# rubikrotation.pl is a Perl-5 program and free software:
# This program can be redistributed and/or modified under the terms
# of the LaTeX Project Public License Distributed from CTAN
# archives in directory macros/latex/base/lppl.txt; either
# version 1 of the License, or any later version.
#
# rubikrotation.pl is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.

##---------------------------------------------------------------------
## OVERVIEW
## This program is part of the rubikrotation package, and is complementary to
## the LaTeX rubikcube package. It processes Rubik rotation sequences  on-the-fly.
## The program reads a datafile (rubikstate.dat) output by the rubikcube package
## and writes the new state to the file rubikstateNEW.dat, which is then input
## by the TeX file. Further documentation  accompanies the rubikrotation package.
```

```
## Note that all possible state changing rotations of a 3x3x3 cube are
## either combinations of, or the inverse of, just 9 different rotations,
## three associated with each XYZ axis.
##-------------------------------------------------------------------
```

3.1 MAIN

```
##==MAIN==
##
## This main module opens three files, and
##      sets up an array for collecting all errors (%error), and sets an error flag to "",
##      reads in the rubik state data file =rubikstate.dat (written by rubikrotation.sty),
##      and calls subs to write the TeX_OUT_FILE,
##      and finally closes all files.
## Each line of the input file (rubikstate.dat) is a comma separated list of arguments.
## The first argument in each line of the file rubikstate.dat is a rubikkeyword.
##
##---------------
## set autoflush for outputs
## $|=1;
##--------------
our $source_file     = "";
our $out_file        = "rubikOUT.txt";                              #default
our $argc            = @ARGV;
our $commandLineArgs = join( " ", @ARGV );
our $showargs        = "\tcommandline args = $commandLineArgs\n";
our $usage =
"\tUsage: rubikrotation [-h|--help|-v|--version] -i <input file> [-o <out file>]\n";
our $rubikversion = "\tVersion: this is rubikrotation version $version\n";
#
## check for correct number of commandline  arguments and allocate filenames
#
if ( $argc == 0 || $argc > 4 ) {    # croak if  0 or  more than 4 arguments
    croak $rubikversion, $showargs, "\tWrong no of arguments\n", $usage;
```

```
}
else {
  SWITCHES:
    while ( $_ = $ARGV[0] ) {
        shift;
        if ( /^-h$/ || /^--help$/ ) {
            die $rubikversion, $usage,
                "\twhere,\n"
              . "\t[-h|--help]\tgives this help listing\n"
              . "\t[-v|--version]\tgives version\n"
              . "\t[-i]     \tcreates specified input file\n",
              "\t[-o]     \tcreates specified output file\n",
              "\tFor documentation see: rubikrotation.pdf,\n",
              "\trubikrotationPL.pdf and rubikcube.pdf.\n\n";
        }
        elsif ( /^-v$/ || /^--version$/ ) { die $rubikversion; }
        elsif (/^-i$/) {
            if ( !@ARGV ) {
                croak $showargs, "\tNo input file specified!\n", $usage;
            }
            else {
                $source_file = $ARGV[0], shift;
            }
        }
        elsif (/^-o$/) {
            if ( !@ARGV ) {
                croak $showargs, "\tNo output file specified!\n", $usage;
            }
            else {
                $out_file = $ARGV[0], shift;
            }
        }
        elsif (/^-\w+/) {
            croak $showargs, "\t$_: Illegal command line switch!\n", $usage;
        }
        else {
```

```
            croak $showargs,
              "\tmissing filenames or ? missing -i or -o switch!\n",
              $usage;
        }
    }    # end of while
}
;        # end of else

#================================

open( IN_FILE, "<$source_file" )
  || croak "\tCan't open source file: $source_file\n";
open( TeX_OUT_FILE, ">$out_file" )
  || croak "\tCan't open output file: $out_file\n";

## create error file (for append)
open( ERROR_OUT_FILE, ">>rubikstateERRORS.dat" )
  || croak "ERROR: can't open file rubikstateERRORS.dat\n";

## use dots for Perl messages (I have used dashes for LaTeX messages in the .sty)
## gprint sub prints its argument (message) to both the screen and to the TeX_OUT_FILE

gprint("");    # newline
gprint("...PERL process.............................");
gprint("...script = rubikrotation.pl $version");

## setup  global error parameters, so we can write all the errors to a file as an array
our %error       = ();    # setup an array for error messages (was %)
our $erroralert  = "";    # error flag
our $errornumber = 0;     #set number of errors to zero

gprint("...reading the current cube state (from File: $source_file)");

our $dataline     = "";
our $newdataline  = "";
our $rubikkeyword = "";
```

```
our $cubesize     = "";  ##  to hold the size, as three (Rubik) or two (twocube)
our $rotationcommand = "";
our @data            = ();

our $Sequence             = "";    ## will hold the original (SHORT) sequence
our $rotationseqNEW       = "";    ## will hold the LONG sequence
our $RotationSequenceName = "";

our $SequenceName  = "";
our $SequenceShort = "";
our $SequenceLong  = "";
our $SequenceInfo  = "";

our $jrcode = 0;    ## We initialise a loop counter for use in the rotation sub
## (see line 624)

#---------inverse mode------------
# a keyword INVERSE or inverse in an infoblock <..>
# FLAG is set (line 400) in response to detecting an infloblock.
#   A set FLAG triggers  (a) reversing rotation sequence (line 484),
#   and (b) inverting each rotation (to generate the inverse sequence).
# Here we define direction FLAG for the INVERSE sequence of rotations.
# The conditional test is in the SUB rotation
our $inverse       = "INV";
our $directionflag = "";

#-------------------------

LINE: while (<IN_FILE>) {
    next LINE if /^#/;    #skip comments
    next LINE if /^%/;    #skip comments
    next LINE if /^$/;    #skip blank lines

    print " \n TOP --------- (new line)\n\n";

    $dataline = $_;       # grab the whole line as a string
```

```
chomp $dataline;      # remove the line-ending character

## clean leading and trailing whitespace
$dataline = cleanstring($dataline);

#check syntax of the string
$rotationcommand = $dataline;    ## needed for error messages
CheckSyntax($dataline);

## form an array so we can process the (rubik)keywords.
@data = split( /,/, $dataline );    # create an array called data

print " dataline array = @data\n";

#-------------------------

## we have 10 fields (0--9)
## check for rubikkeyword= cubesize, up,down,left,right,front,back,checkstate,rotation:
$rubikkeyword = $data[0];

##---------------------------------
## RWDN 2 August 2017
## introduced keyword cubesize so prog can distinguish
##    between a TWOcube and an THREEcube.
## Here we check for the rubikkeyword 'cubesize'
## cubesize is currently only being used to change the array size in random SUB
if ( $rubikkeyword eq 'cubesize' ) {
    gprint("...");
    $rotationcommand = $dataline;    ## used in output message
    gprint("...command = $rotationcommand");
    $cubesize = RemoveAllSpaces( $data[1] );
    if ( $cubesize eq "two" )   { gprint("...cube = TWOcube") }
    if ( $cubesize eq "three" ) { gprint("...cube = THREEcube") }
    gprint("...");
    next LINE;
}
```

```
## ----------------------------------

## RWDN 7 February 2018
## we vary the number of leading dots for the gprint command
##    so as to make the array of colour codes (X,W,Y, etc) form
##    a nice square when printed to the log file (the standard no is 3 dots)

if ( $rubikkeyword eq 'up' ) {
    gprint(".....$dataline");
    $Ult[0]   = $data[1], $Umt[0] = $data[2], $Urt[0] = $data[3],
      $Ulm[0] = $data[4], $Umm[0] = $data[5], $Urm[0] = $data[6],
      $Ulb[0] = $data[7], $Umb[0] = $data[8], $Urb[0] = $data[9];
    next LINE;
}

if ( $rubikkeyword eq 'down' ) {
    gprint("....$dataline");
    $Dlt[0]   = $data[1], $Dmt[0] = $data[2], $Drt[0] = $data[3],
      $Dlm[0] = $data[4], $Dmm[0] = $data[5], $Drm[0] = $data[6],
      $Dlb[0] = $data[7], $Dmb[0] = $data[8], $Drb[0] = $data[9];
    next LINE;
}

if ( $rubikkeyword eq 'left' ) {
    gprint("....$dataline");
    $Llt[0]   = $data[1], $Lmt[0] = $data[2], $Lrt[0] = $data[3],
      $Llm[0] = $data[4], $Lmm[0] = $data[5], $Lrm[0] = $data[6],
      $Llb[0] = $data[7], $Lmb[0] = $data[8], $Lrb[0] = $data[9];
    next LINE;
}

if ( $rubikkeyword eq 'right' ) {
    gprint("...$dataline");
    $Rlt[0]   = $data[1], $Rmt[0] = $data[2], $Rrt[0] = $data[3],
      $Rlm[0] = $data[4], $Rmm[0] = $data[5], $Rrm[0] = $data[6],
      $Rlb[0] = $data[7], $Rmb[0] = $data[8], $Rrb[0] = $data[9];
```

```
    next LINE;
}

if ( $rubikkeyword eq 'front' ) {
    gprint("...$dataline");
    $Flt[0]   = $data[1], $Fmt[0] = $data[2], $Frt[0] = $data[3],
      $Flm[0] = $data[4], $Fmm[0] = $data[5], $Frm[0] = $data[6],
      $Flb[0] = $data[7], $Fmb[0] = $data[8], $Frb[0] = $data[9];
    next LINE;
}

if ( $rubikkeyword eq 'back' ) {
    gprint("....$dataline");
    $Blt[0]   = $data[1], $Bmt[0] = $data[2], $Brt[0] = $data[3],
      $Blm[0] = $data[4], $Bmm[0] = $data[5], $Brm[0] = $data[6],
      $Blb[0] = $data[7], $Bmb[0] = $data[8], $Brb[0] = $data[9];
    next LINE;
}

## if the rubikkeyword is 'checkstate'
##    we just check the  state and write the output data to a file.
if ( $rubikkeyword eq 'checkstate' ) {
    gprint("...");
    $rotationcommand = $dataline;    ## used in output message
    gprint("...command = $rotationcommand");
    checkstate();
    next LINE;
}

## IF the rubikkeyword is 'rotation'
##   we first check to see  if the second argument=random.
##   ---if so, then we check that the third argument is an integer,
##   ---if it is an integer n --> random  => random(n)
## ELSE   it  must be a rotation sequence  --> send elements to rotation sub.

if ( $rubikkeyword eq 'rotation' )
```

```
{    ## this IF runs down to near end of MAIN

    ##RWDN 10 Aug 2017
    ## moved checkstate  to be inside rotation (so a next LINE will terminate prog)
    gprint("...");
    gprint("...rotation keyword");
    checkstate();

    gprint("...")
      ;    ## logfile marker for begining of 'rotation/random' process

    # we now grab a copy of the dataline, and we shall use this
    # in the ErrorMessage SUB  to indicate which command
    # an error is in.
    $rotationcommand = $dataline;    ## used in output message
    gprint("...command = $rotationcommand");

    # need to check that a second argument exists (else --> ErrorMessage).
    #  ---should be either 'random',
    #  ---or a macroname for a rotation sequence,
    #  ---or the first element of a rotation sequence.

    if ( $data[1] eq "" ) {    # no second argument
        gprint("..*missing second argument");
        ErrorMessage("QUITTING PERL PROGRAM --- missing second argument:");
        ErrorMessage("--- ? bad rotation macro-name");
        quitprogram();
    }

    ##---------keyword = random-----------------
    ##  (command used for scrambling the cube)
    ## if second argument in $dataline = random
    ##   THEN we also need to check if third argument is an integer;
    ## if so  send integer --> random sub.
```

```
##----------------------------
## (28 July 2017: RWDN) : bugfix:
##  better syntax checking required for the <random,n> command
##  as spaces  before or after commas  caused errors.
##----------------------------

## allow upper and lowercase keyword random
if ( lc( $data[1] ) =~ m/random/ ) {
    ## the string contains the keyword  random

    ## now check for missing comma after the keyword
    if ( lc( RemoveAllSpaces( $data[1] ) ) ne "random" )
    {    ## error, ? missing comma
        ErrorMessage("[$data[1]] --- missing comma after ‘random’ ");
        next LINE;
    }

    ## now check for the trailing integer

    if (    ( lc( RemoveAllSpaces( $data[1] ) ) eq "random" )
        and ( $data[2] eq "" ) )
    {
        ## missing integer
        ErrorMessage("[$data[2]] --- missing integer after ‘random,’");
        next LINE;
    }

    if ( RemoveAllSpaces( $data[2] ) =~ /\D/ ) {
        ##  Note that the \D  operator sees ,  23,  as a word not an integer.
        ## so if true then cannot be a number (D matches word and space elements)
        ErrorMessage("[$data[2]] --- this is not an integer");
        next LINE;
    }
    else {    ## string consists of one or more integers
        ## check to see if more than one integer exists
        ## by seeing if the string changes if we remove all the spaces
```

```
                ## (note we have to use a string with the m operator)
                my $RAS = RemoveAllSpaces( $data[2] );
                if ( $data[2] =~ m/$RAS/ ) {
                    ## OK so this must be a single integer
                    ## so we can now do n random rotations
                    ## by sending the integer to the random SUB
                    random( $data[2] );
                    next LINE;
                }
                else {    ## there must be spaces separating several integers;
                    ErrorMessage("[$data[2]] --- only one integer allowed");
                    next LINE;
                }
            }
            ;             ## end of else
        } ## end of IF
        ##-------------------------------------------------------------

        else {
            ## -----rotation sequence---------
            ## the line must be a rotation sequence line, so send the sequence
            # to the rotation sub;

            # Note that a copy of the rotation command is already held in the
            # variable rotationcommand (see above). It is used in the
            # ErrorMessage SUB.

            #----<infoblocks>---------------------------

# infoblocks are strings bounded by angle brackets <..>
# and are designed for holding metadata.
#
# Multiple comma separated infoblocks are allowed (but NOT nested).
# All infoblocks are eventually concaternated into a colon separated string, and
# returned into the OUT file (= rubikstateNEW.dat) as the macro \SequenceInfo.
#
```

```
# We process and then remove any infoblocks which exist.
#  infoblocks  are chars delimited by <...>
#
# The SUB infoblockcolon replaces any commas with a colon (so as to
#   facilitate  string manipulation, and allows us to distinguish between
#   a string and a data array), and  returns a new string (= $newdataline).
#
# The RubikRotation argument  allows <infoblocks>   for carrying  special
#    keywords, eg <inverse> which can be used to influence the process.
# If several infoblocks exist, then we collect the contents into
#   variable SequenceInfo, and separate them with a colon;
#
# The SUB cutinfoblock  returns TWO strings:
#  (1) the name of the new revised string = newdataline, (with infoblocks removed)
#  (2) the contents of the infoblock = $SequenceInfo

            infoblockcolon($dataline);

            ## rename the returned newdataline string  to dataline
            ## and reinitialise the string  newdataline so it can be used again.
            $dataline    = $newdataline;
            $newdataline = "";             ## reset the variable

            gprint("...dataline = $dataline");

            ## now pass the string to cutinfoblock
            local @seq = ();

            while ( ( index $dataline, '<' ) != -1 ) {

                cutinfoblock($dataline);

               # best to use the whole word <inverse> to avoid errors
               # best to force lowercase so users can type the word as they want

                if ( lc($SequenceInfo) =~ m/(inverse)/ ) {
```

```
        ## set a FLAG
        $directionflag = $inverse;
        print " FLAG set to = $inverse\n";
    }

    # append each infoblock to an array
    push @seq, $SequenceInfo;
    $dataline = $newdataline;
}

# finally, we join the seqInfo array into a string so we can print it
$SequenceInfo = join( "; ", @seq );

#--------repeat blocks--------------------------

## there are now no more infoblocks, so we now look for repeat-blocks.
## these are embedded inside the rotation sequence

## we first reformulate any repeat blocks (,) --> {;} if they exist
## this is to allow us to process any repeat blocks as separate elements
## so we look for curved brackets ie indicating a repeat block, and
## if we find a ( we then send the dataline to the SUB fixrepeatelement()
## the  SUB fixrepeatelement() then  returns the new revised  dataline string
## containing the FIRST repeat block which has been expanded.
## If there is another ( then we repeat the procedure until all
## repeat blocks have been expanded, and incorporated into the mail rotation string.

while ( ( index $dataline, '(' ) != -1 ) {
    fixrepeatelement($dataline);
    $dataline    = $newdataline;
    $newdataline = "";             ## reset the variable
}

## rename remaining dataline string as SequenceShortBrace
## since if there are any repeat blocks, they are now reformulated with braces and semicolons
## ie (,) --> {;} etc
```

```
      $SequenceShortBrace = $dataline;

      ## clean leading and trailing whitespace
      $SequenceShortBrace = cleanstring($SequenceShortBrace);

      ##----------------
      ## form a new array from $SequenceShortBrace (since we have changed the format
      ## slightly; ie some commands may have been reformulated as semicolons).
      @data = split( /,/, $SequenceShortBrace );

      ## need to remove keyword <rotation> (= first element in the array)
      ## removing it late like this is convenient for error checking purposes,
      ## as then the keyword 'rotation' is on the string
      shift(@data);

      ## now need to recreate the  string from the array @data for use later
      ## (as rotation keyword has been removed)

      $SequenceShortBrace = join( ",", @data );

      #-----create SequenceShort, so we can output it later----

#   since the 'rotating' keyword has been removed from the string,
#   we can replace (repair to original state) any braces  or or semicolons
#   around repeat strings (if exist) and then rename it as SequenceShort
#   which we will output at the end (in SUB writestate).

      if ( ( index $SequenceShortBrace, '{' ) != -1 ) {
          print " repairing braces and semicolon--> ()\n";
          ## swap: BBook p 138--139
          $SequenceShortBrace =~ tr/\{/(/;    # swap { --> (
          $SequenceShortBrace =~ tr/\}/)/;    # swap } --> )
          $SequenceShortBrace =~ tr/;/,/;     # swap ; --> ,
      }

      #rename to SequenceShort
```

```
$SequenceShort = $SequenceShortBrace;

print " SequenceShort = $SequenceShort\n";
##---------------------------

## now we continue processing the array  "data"

my $n = 0;    ##total no of array elements in "data"
$n = ( $#data + 1 );
print " processing rotation arguments: = @data (n= $n)\n";

## --------check for state of  direction flag----------------------
##  FLAG defined in line 224.
## FLAG is set in line 400 on detecting <..> delimeters = infoblock
## if  flag set (by inverse keyword) then reverse the sequence array
if ( $directionflag eq $inverse ) {

    # FLAG is set, so we need to inverse the array
    gprint("...directionFLAG set; reversing array...");

    # but before reversing, look at the first array element
    # to see if it is a square bracket element = NAME element
    # so check the first char to see if it is [
    if ( substr( $data[0], 0, 1 ) eq '[' ) {
        $SequenceName = $data[0];
        print " SequenceName (inv) =  $SequenceName \n";
    }

    @data = reverse @data;
    print " processing rotation arguments: = @data (n= $n)\n";
}

# send each rotation element to the sub rotation()
print " CALLing SUB rotation\n";

foreach $element (@data) {
```

```
                ## clean leading and trailing white space
                $element = cleanstring($element);
                ## send element to rotation SUB
                rotation($element);
            }

        }    # end of else
    }
    ;        # end of IF ( re: rotation keyword)

    #---------------------------------------------
    ## place any new keywords for processing here
    ##---------------------------------------------

};    ## end of while

## we have now finished reading in all the lines from the source file,
## and processing all the rotations etc,
## so we now just write the new cube state
## to the output file = TeX_OUT_FILE (so LaTeX can read it)
## plus any ErrorMessages
## -- all these are handled by the quitprogram sub

quitprogram();

##==============end of main==========================
```

3.2 rotation

```
sub rotation {

    print " SUB rotation\n";

    ## here we process the array @data  (from main) consisting of all
    ## the rotation commands associated with
```

```
## a single RubikRotation command -- the ‘rotation’ key word has already been removed
##  so  we start  here with [name] if it exists.

##---------------------------
## variables used in SUBs  rotation() and rubikmod()
## need to be defined outside the SUBs

$modnumber = -1;    #multiple associated with the char, eg D2 etc
$rotcode   = "";
$rotnumber = 0;

#------------------
my @repeatcode = ();

my $m             = -1;
my $originalrcode = "";
my $j;                ## used with m below
my $numberofchars;    ## length of a string
my $nfrontchars;

##-----------------

## grab the rotation code passed to this sub from MAIN
my $rcode = $_[0];

## now we start a big loop processing each cs-element (= rcode),
## and collecting these elements into two cs-strings
## ($Sequence --> original string output as SHORT string (has codes like R2,L3 etc),
##  and $rotationseqNEW --> output as LONG string -- all short codes expanded)

## first, clean leading and trailing white space (eg between,  R ,)
$rcode = cleanstring($rcode);

## grab a copy of the element (char) for use if  m Mod4=0
$originalrcode = $rcode;
```

```
## increment the LOOP counter
## (initialised  using <our> in MAIN = line 226)
## for use in the rotation SUB.
## This counter is used to identify the first element (rcode)
## and used to grab [name] --> SequenceName.
$jrcode = $jrcode + 1;    ## increment rotation element (char) counter

## -----check for [nameblocks]-----------------------------
##
## We look at the first character of each element in the sequence
## if an element has a leading [  then it is a label (not a rotation)
## If this is the case, then jump to next element in the array

## BUT if trailing comma is missing, then (error as  next rotation will be included
##    as part of the label) so need to trap this and
## make the test: is first AND last char a sq bracket?
## (strictly only need to look at /first/ char, as the early syntax check will have
##     detected any unbalanced brackets already)

if (    ( substr( $rcode, 0, 1 ) =~ /\[/ )
    and ( substr( $rcode, -1 ) ) =~ /\]/ )
{

    gprint("...$rcode is a label OK");

    if ( $directionflag eq $inverse ) {

        # do nothing
    }

    else {
        ## if this 'label' is also the FIRST element, then label = nameblock
        if ( $jrcode == 1 ) { $SequenceName = $rcode }
    }
    ;    # end of IF
```

```
    ## now get next rotation element
    next;
}
;       ## end of if

##--------------

## the rcode must therefore be either a rotation code or a repeat-block.

##-------check for (repeatblocks)---------------------------------
##
## we have already replaced any repeat chars (,) with {;}
## so we now check for  elements with leading { and then expand  them
##   the appropriate number of times.
## Note that the actual expansion is done by SUB repeat()

## Note that if there is NO comma before the {} of a {repeat block}, then
## the true repeat block  will not be recognised by the
## usual test -- since the test is looking for a leading {  etc.
## However, in this event, the string being handled (not a true element)
## will be processed  as if it were a rotation, and an
## error will be thrown, so it will get picked up OK.

if ( substr( $rcode, 0, 1 ) =~ /\{/ ) {
    print " repeat block found        = $rcode \n";

    ## since we now want to send each rotation element in the repeat block to
    ## the rotation sub, we need to replace any ; with commas
    ## therefore translate ; --> , but retain the {}
    $rcode =~ tr/;/,/;

    print " repeat block reformulated = $rcode \n";

    #-------log file message--------
    ## log file: we want to show the repeat string  in the users original form
    ## so we translate it back to the user's  orig form {,} --> (,)
```

```
$origrcode = $rcode;
$origrcode =~ tr/\{/(/;
$origrcode =~ tr/\}/)/;
gprint("...Expanding: $origrcode ...");

#-------------

#-----------
## expand the code in the repeat block
print " CALLing SUB: repeat($rcode)\n";

repeat($rcode);    # this expands the repeated elements in the block

## this sub returns the expanded form as $insert
$expanded_repeatcode = $insert;

print " expanded_repeatcode = $expanded_repeatcode\n\n";

#------------

#----------------
# process each new element in the expanded_repeatcode --> rotation
# make expanded_repeatcode into an array, and send each element on

@repeatcode = split( /,/, $expanded_repeatcode );

## -----------check for direction flag-----------------
## if flag set then reverse the array
if ( $directionflag eq $inverse ) { @repeatcode = reverse @repeatcode }

# send each element to rotation SUB for processing
print " CALLing SUB rotation\n";
foreach $E (@repeatcode) {
    print " sending repeat element $E to rotation SUB\n";
    rotation($E);
}
```

```
    # when this foreach is finished, then get next rotation element from
    # the original @data array (see  foreach.. near end of  MAIN)
    next;

}
;    ## end of if

##================================

## if an element has got this far,  it must be a single rotation code
## (maybe with a trailing digit), so it needs processing  as a rotation
## and appending the code to what will become the SequenceLONG string.

##-----------------------

## CALL the sub rubikmod to process the rotation element,
##   and to return  the front code (= $rotcode), orig no = $rotnumber,
##   and mod4 value (= $modnumber).

rubikmod($rcode);

## update rcode  <--- rotcode   (returned by the SUB rubikmod() )
## collect   $m  <--- modnumber (returned by the SUB rubikmod() )
$rcode = $rotcode;
$m     = $modnumber;

## we collect all the new versions of rcode into a cs-string = $SequenceLong
##  which will finally be output as the LONG string

#-----------------------
# check with directionflag
if ( $directionflag eq $inverse ) { $rcode = inverse($rcode) }

#------------------------
```

```
    ##--------------------
    if ( $m == 0 ) {
        ## do NOT implement the rotation code, and do NOT append to SequenceLong
        ## print the /original/ rcode (eg R4, or D0 etc)
        gprint(
"..*rotation ,$originalrcode, ERROR  ($rotnumber = 0  mod 4) not implemented"
        );
        ErrorMessage(
            ",$originalrcode,  -- ($rotnumber = 0  mod 4) not implemented");
        next;
    }

    if ( $m == 1 ) {
        if ( $rotnumber >= 5 ) {
            gprint("...Expanding $originalrcode ($rotnumber = $m  mod 4) ...");
        }
        $SequenceLong = $SequenceLong . $rcode . ",";
    }
    else {
        # m =  2 or 3
        if ( $rotnumber >= 5 ) {
            gprint("...Expanding $originalrcode ($rotnumber = $m  mod 4) ...");
        }
        else { gprint("...Expanding $originalrcode ...") }

        for ( $j = 1 ; $j <= $m ; $j++ )
        {    ## append  rcode  m times  to sequenceLONG
            $SequenceLong = $SequenceLong . $rcode . ",";
        }
    }
    ;        ## end of else

    ##--------------------

    ## if single trailing digit present,
    ##     then we implement the rotation command m times.
```

```
    ## if more than one trailing digit
    ##     then the error is trapped at the end (as frontstring will not be recognised
    ##     ie will not be  in the following list, and hence will be trapped as an error, eg R3)

##-----------------------
## RWDN Sept29 2017 testing to fix  WESTERN notation problem
##  arrange for user to be able to include <western> in metadata etc
##  -- as this will mean that one can store these algorithms as a macro

    if ( lc($SequenceInfo) =~ m/(western)/ ) {

        if ( $rcode eq "l" ) {
            $rcode = "Lw";
            gprint("...WESTERN NOTATION: rotation l  --> Lw,    OK");
        }
        if ( $rcode eq "lp" ) {
            $rcode = "Lwp";
            gprint("...WESTERN NOTATION: rotation lp --> Lwp,   OK");
        }
        if ( $rcode eq "r" ) {
            $rcode = "Rw";
            gprint("...WESTERN NOTATION: rotation r  --> Rw,    OK");
        }
        if ( $rcode eq "rp" ) {
            $rcode = "Rwp";
            gprint("...WESTERN NOTATION: rotation rp --> Rwp,   OK");
        }
        if ( $rcode eq "f" ) {
            $rcode = "Fw";
            gprint("...WESTERN NOTATION: rotation f  --> Fw,    OK");
        }
        if ( $rcode eq "fp" ) {
            $rcode = "Fwp";
            gprint("...WESTERN NOTATION: rotation fp --> Fwp,   OK");
        }
        if ( $rcode eq "b" ) {
```

```
            $rcode = "Bw";
            gprint("...WESTERN NOTATION: rotation b  --> Bw,    OK");
        }
        if ( $rcode eq "bp" ) {
            $rcode = "Bwp";
            gprint("...WESTERN NOTATION: rotation bp --> Bwp,   OK");
        }
        if ( $rcode eq "u" ) {
            $rcode = "Uw";
            gprint("...WESTERN NOTATION: rotation u  --> Uw,    OK");
        }
        if ( $rcode eq "up" ) {
            $rcode = "Uwp";
            gprint("...WESTERN NOTATION: rotation up --> Uwp,   OK");
        }
        if ( $rcode eq "d" ) {
            $rcode = "Dw";
            gprint("...WESTERN NOTATION: rotation d  --> Dw,    OK");
        }
        if ( $rcode eq "dp" ) {
            $rcode = "Dwp";
            gprint("...WESTERN NOTATION: rotation dp --> Dwp,   OK");
        }

    }
##----------------------

    if ( $rcode eq "L" ) {
        for ( $j = 1 ; $j <= $m ; $j++ ) {
            gprint("...rotation L,   OK (= Lp3)");
            &rrL;
        }
    }
    elsif ( $rcode eq "Lp" ) {
        for ( $j = 1 ; $j <= $m ; $j++ ) {
            gprint("...rotation Lp,  OK");
```

```
        &rrLp;
    }
}
elsif ( $rcode eq "Lw" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Lw,  OK (= Lp3 + Srp)");
        &rrLw;
    }
}
elsif ( $rcode eq "Lwp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Lwp, OK (= Lp + Sr)");
        &rrLwp;
    }
}
elsif ( $rcode eq "Ls" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Ls,  OK (= L + Rp)");
        &rrLs;
    }
}
elsif ( $rcode eq "Lsp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Lsp, OK (= Lp + R)");
        &rrLsp;
    }
}
elsif ( $rcode eq "La" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation La,  OK (= L + R)");
        &rrLa;
    }
}
elsif ( $rcode eq "Lap" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Lap, OK (= Lp + Rp)");
```

```
        &rrLap;
    }
}
####
elsif ( $rcode eq "R" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) { gprint("...rotation R,   OK"); &rrR }
}
elsif ( $rcode eq "Rp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Rp,  OK (= R3)");
        &rrRp;
    }
}
elsif ( $rcode eq "Rw" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Rw,  OK (= R + Sr)");
        &rrRw;
    }
}
elsif ( $rcode eq "Rwp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Rwp, OK (= Rp + Srp)");
        &rrRwp;
    }
}
elsif ( $rcode eq "Rs" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Rs,  OK (= R + Lp)");
        &rrRs;
    }
}
elsif ( $rcode eq "Rsp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Rsp, OK (= Rp + L)");
        &rrRsp;
    }
```

```
}
elsif ( $rcode eq "Ra" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Ra,  OK (= R + L)");
        &rrRa;
    }
}
elsif ( $rcode eq "Rap" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Rap, OK (= Rp + Lp)");
        &rrRap;
    }
}
####
elsif ( $rcode eq "U" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) { gprint("...rotation U,   OK"); &rrU }
}
elsif ( $rcode eq "Up" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Up,  OK (= U3)");
        &rrUp;
    }
}
elsif ( $rcode eq "Uw" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Uw,  OK (= U + Su)");
        &rrUw;
    }
}
elsif ( $rcode eq "Uwp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Uwp, OK (= Up + Sup)");
        &rrUwp;
    }
}
elsif ( $rcode eq "Us" ) {
```

```
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Us,  OK (= U + Dp)");
        &rrUs;
    }
}
elsif ( $rcode eq "Usp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Usp, OK (= Up + D)");
        &rrUsp;
    }
}
elsif ( $rcode eq "Ua" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Ua,  OK (= U + D)");
        &rrUa;
    }
}
elsif ( $rcode eq "Uap" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Uap, OK (= Up + Dp)");
        &rrUap;
    }
}
####
elsif ( $rcode eq "D" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation D,   OK (= Dp3)");
        &rrD;
    }
}
elsif ( $rcode eq "Dp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Dp,  OK ");
        &rrDp;
    }
}
```

```
elsif ( $rcode eq "Dw" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Dw,  OK (= Dp3 + Sup)");
        &rrDw;
    }
}
elsif ( $rcode eq "Dwp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Dwp, OK (= Dp + Su)");
        &rrDwp;
    }
}
elsif ( $rcode eq "Ds" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Ds,  OK (= D + Up)");
        &rrDs;
    }
}
elsif ( $rcode eq "Dsp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Dsp, OK (= Dp + U)");
        &rrDsp;
    }
}
elsif ( $rcode eq "Da" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Da,  OK (= D + U)");
        &rrDa;
    }
}
elsif ( $rcode eq "Dap" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Dap, OK (= Dp + Up)");
        &rrDap;
    }
}
```

```
####
elsif ( $rcode eq "F" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) { gprint("...rotation F,   OK"); &rrF }
}
elsif ( $rcode eq "Fp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Fp,  OK (= F3)");
        &rrFp;
    }
}
elsif ( $rcode eq "Fw" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Fw,  OK (= F + Sf)");
        &rrFw;
    }
}
elsif ( $rcode eq "Fwp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Fwp, OK (= Fp + Sfp)");
        &rrFwp;
    }
}
elsif ( $rcode eq "Fs" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Fs,  OK (= F + Bp)");
        &rrFs;
    }
}
elsif ( $rcode eq "Fsp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Fsp, OK (= Fp + B)");
        &rrFsp;
    }
}
elsif ( $rcode eq "Fa" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
```

```
        gprint("...rotation Fa,  OK (= F + B)");
        &rrFa;
    }
}
elsif ( $rcode eq "Fap" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Fap, OK (= Fp + Bp)");
        &rrFap;
    }
}
####
elsif ( $rcode eq "B" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation B,   OK (= Fp3)");
        &rrB;
    }
}
elsif ( $rcode eq "Bp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Bp,  OK");
        &rrBp;
    }
}
elsif ( $rcode eq "Bw" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Bw,  OK (= Fp3 + Sfp)");
        &rrBw;
    }
}
elsif ( $rcode eq "Bwp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Bwp, OK (= Fp + Sf)");
        &rrBwp;
    }
}
elsif ( $rcode eq "Bs" ) {
```

```
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Bs,  OK (= B + Fp)");
        &rrBs;
    }
}
elsif ( $rcode eq "Bsp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Bsp, OK (= Bp + F)");
        &rrBsp;
    }
}
elsif ( $rcode eq "Ba" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Ba,  OK (= B + F)");
        &rrBa;
    }
}
elsif ( $rcode eq "Bap" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Bap, OK (= Bp + Fp)");
        &rrBap;
    }
}

#### -------------------------------------

#### inner-slice (= middle slice)
## need to include MES (middle slice) notation
elsif ( $rcode eq "M" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation M,   OK (= Sl) ");
        &rrSl;
    }
}
elsif ( $rcode eq "Mp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
```

```
        gprint("...rotation Mp,  OK (= Sr) ");
        &rrSr;
    }
}
elsif ( $rcode eq "E" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation E,   OK (= Sd) ");
        &rrSd;
    }
}
elsif ( $rcode eq "Ep" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Ep,  OK (= Su) ");
        &rrSu;
    }
}
elsif ( $rcode eq "S" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation S,   OK (= Sf) ");
        &rrSf;
    }
}
elsif ( $rcode eq "Sp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Sp,  OK (= Sb) ");
        &rrSb;
    }
}

#### middle slice rotations (Singmaster)
elsif ( $rcode eq "Su" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Su,  OK ");
        &rrSu;
    }
}
```

```
elsif ( $rcode eq "Sup" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Sup, OK (= Su3)");
        &rrSup;
    }
}
elsif ( $rcode eq "Sd" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Sd,  OK (= Sup)");
        &rrSd;
    }
}
elsif ( $rcode eq "Sdp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Sdp, OK (= Su)");
        &rrSdp;
    }
}
elsif ( $rcode eq "Sl" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Sl,  OK (= Srp)");
        &rrSl;
    }
}
elsif ( $rcode eq "Slp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Slp, OK (= Sr)");
        &rrSlp;
    }
}
elsif ( $rcode eq "Sr" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Sr,  OK");
        &rrSr;
    }
}
```

```
elsif ( $rcode eq "Srp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Srp, OK (= Sr3)");
        &rrSrp;
    }
}
elsif ( $rcode eq "Sf" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Sf,  OK");
        &rrSf;
    }
}
elsif ( $rcode eq "Sfp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Sfp, OK (= Sf3)");
        &rrSfp;
    }
}
elsif ( $rcode eq "Sb" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Sb,  OK (= Sfp)");
        &rrSb;
    }
}
elsif ( $rcode eq "Sbp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation Sbp, OK (= Sf)");
        &rrSbp;
    }
}

## need to include Jaap Puzzles website for middle slice notation (Lm, Lmp)
## also include Randelshofer website middle slice notation (ML,MLp..)

elsif ($rcode eq "ML"
    or $rcode eq "MRp"
```

```
    or $rcode eq "Lm"
    or $rcode eq "Rmp" )
{
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation $rcode  OK (= Lm = M = Sl) ");
        &rrSl;
    }
}

elsif ($rcode eq "MR"
    or $rcode eq "MLp"
    or $rcode eq "Rm"
    or $rcode eq "Lmp" )
{
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation $rcode  OK (= Rm = Mp = Sr) ");
        &rrSr;
    }
}

elsif ($rcode eq "MU"
    or $rcode eq "MDp"
    or $rcode eq "Um"
    or $rcode eq "Dmp" )
{
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation $rcode  OK (= Um = Ep = Su) ");
        &rrSu;
    }
}

elsif ($rcode eq "MD"
    or $rcode eq "MUp"
    or $rcode eq "Dm"
    or $rcode eq "Ump" )
{
```

```
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation $rcode  OK (= Dm = E = Sd) ");
        &rrSd;
    }
}

elsif ($rcode eq "MF"
    or $rcode eq "MBp"
    or $rcode eq "Fm"
    or $rcode eq "Bmp" )
{
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation $rcode  OK (= Fm = S = Sf) ");
        &rrSf;
    }
}

elsif ($rcode eq "MB"
    or $rcode eq "MFp"
    or $rcode eq "Bm"
    or $rcode eq "Fmp" )
{
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation $rcode  OK (= Bm = Sp = Sb) ");
        &rrSb;
    }
}
##---------------------------------

#### double outer slice (wide) notation
#### need to include Randelshofer TL, TLp double outer slice notation
#### (equiv to the  w wide notation)
elsif ( $rcode eq "TL" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation TL,  OK (= Lw = Lp3 + Srp)");
        &rrLw;
```

```
    }
}
elsif ( $rcode eq "TLp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation TLp, OK (= Lwp = Lp + Sr)");
        &rrLwp;
    }
}
elsif ( $rcode eq "TR" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation TR,  OK (= Rw = R + Sr)");
        &rrRw;
    }
}
elsif ( $rcode eq "TRp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation TRp, OK (= Rwp = Rp + Srp)");
        &rrRwp;
    }
}
elsif ( $rcode eq "TU" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation TU,  OK (= Uw = U + Su)");
        &rrUw;
    }
}
elsif ( $rcode eq "TUp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation TUp, OK (= Uwp = Up + Sup)");
        &rrUwp;
    }
}
elsif ( $rcode eq "TD" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation TD,  OK (= Dw = Dp3 + Sup)");
        &rrDw;
```

```
    }
}
elsif ( $rcode eq "TDp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation TDp, OK (= Dwp = Dp + Su)");
        &rrDwp;
    }
}
elsif ( $rcode eq "TF" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation TF,  OK (= Fw = F + Sf)");
        &rrFw;
    }
}
elsif ( $rcode eq "TFp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation TFp, OK (= Fwp = Fp + Sfp)");
        &rrFwp;
    }
}
elsif ( $rcode eq "TB" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation TB,  OK (= Bw = Fp3 + Sfp)");
        &rrBw;
    }
}
elsif ( $rcode eq "TBp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation TBp, OK (= Bwp = Fp + Sf)");
        &rrBwp;
    }
}

## ---------------------------
## opposite slice notation of Randelshofer (SR, SRp)  (= standard Rs, Rsp)
## opposite outer slices rotated in SAME direction as the FACE
```

```
elsif ( $rcode eq "SL" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation $rcode,  OK (= Ls = L + Rp)");
        &rrLs;
    }
}
elsif ( $rcode eq "SLp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation $rcode,  OK (= Lsp = Lp + R)");
        &rrLsp;
    }
}

elsif ( $rcode eq "SR" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation $rcode,  OK (= Rs = R + Lp)");
        &rrRs;
    }
}
elsif ( $rcode eq "SRp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation $rcode,  OK (= Rsp = Rp + L)");
        &rrRsp;
    }
}

elsif ( $rcode eq "SU" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation $rcode,  OK (= Us = U + Dp)");
        &rrUs;
    }
}
elsif ( $rcode eq "SUp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation $rcode,  OK (= Usp = Up + D)");
```

```
        &rrUsp;
    }
}

elsif ( $rcode eq "SD" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation $rcode,  OK (= Ds = D + Up)");
        &rrDs;
    }
}
elsif ( $rcode eq "SDp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation $rcode,  OK (= Dsp = Dp + U)");
        &rrDsp;
    }
}

elsif ( $rcode eq "SF" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation $rcode,  OK (= Fs = F + Bp)");
        &rrFs;
    }
}
elsif ( $rcode eq "SFp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation $rcode,  OK (= Fsp = Fp + B)");
        &rrFsp;
    }
}

elsif ( $rcode eq "SB" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation $rcode,  OK (= Bs = B + Fp)");
        &rrBs;
    }
}
```

```
elsif ( $rcode eq "SBp" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation $rcode,  OK (= Bsp = Bp + F)");
        &rrBsp;
    }
}

## -------------------------

## whole cube rotations
## need to include x,y,z (upper and lowercase) and also u,d,l,r,f,b (lowercase only) equivalents
elsif ( $rcode eq "X" or $rcode eq "x" or $rcode eq "r" or $rcode eq "lp" )
{
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation $rcode, OK (= x = R + Sr + Lp)");
        &rrR;
        &rrSr;
        &rrLp;
    }
}
elsif ($rcode eq "Xp"
    or $rcode eq "xp"
    or $rcode eq "l"
    or $rcode eq "rp" )
{
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation $rcode, OK (= xp = Rp + Srp + L)");
        &rrRp;
        &rrSrp;
        &rrL;
    }
}
elsif ( $rcode eq "Y" or $rcode eq "y" or $rcode eq "u" or $rcode eq "dp" )
{
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation $rcode, OK (= y = U + Su + Dp)");
```

```
        &rrU;
        &rrSu;
        &rrDp;
    }
}
elsif ($rcode eq "Yp"
    or $rcode eq "yp"
    or $rcode eq "d"
    or $rcode eq "up" )
{
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation $rcode, OK (= yp = Up + Sup + D)");
        &rrUp;
        &rrSup;
        &rrD;
    }
}
elsif ( $rcode eq "Z" or $rcode eq "z" or $rcode eq "f" or $rcode eq "bp" )
{
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation $rcode, OK (= z = F + Sf + Bp)");
        &rrF;
        &rrSf;
        &rrBp;
    }
}
elsif ($rcode eq "Zp"
    or $rcode eq "zp"
    or $rcode eq "b"
    or $rcode eq "fp" )
{
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation $rcode, OK (= zp = Fp + Sfp + B)");
        &rrFp;
        &rrSfp;
        &rrB;
```

```
    }
}

## more whole cube notation
## need to include Jaap website whole cube  Lc notation
## also include Randelshofer C notation (CL, CLp.)

elsif ($rcode eq "CL"
    or $rcode eq "CRp"
    or $rcode eq "Lc"
    or $rcode eq "Rcp" )
{
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation $rcode OK (= Lc = xp = Rp + Srp + L)");
        &rrRp;
        &rrSrp;
        &rrL;
    }
}

elsif ($rcode eq "CR"
    or $rcode eq "CLp"
    or $rcode eq "Rc"
    or $rcode eq "Lcp" )
{
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation $rcode OK (= Rc = x = R + Sr + Lp)");
        &rrR;
        &rrSr;
        &rrLp;
    }
}

elsif ($rcode eq "CU"
    or $rcode eq "CDp"
    or $rcode eq "Uc"
```

```
    or $rcode eq "Dcp" )
{
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation $rcode OK (= Uc = y = U + Su + Dp)");
        &rrU;
        &rrSu;
        &rrDp;
    }
}

elsif ($rcode eq "CD"
    or $rcode eq "CUp"
    or $rcode eq "Dc"
    or $rcode eq "Ucp" )
{
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation $rcode OK (= Dc = yp = Up + Sup + D)");
        &rrUp;
        &rrSup;
        &rrD;
    }
}

elsif ($rcode eq "CF"
    or $rcode eq "CBp"
    or $rcode eq "Fc"
    or $rcode eq "Bcp" )
{
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation $rcode OK (= Fc = z = F + Sf + Bp)");
        &rrF;
        &rrSf;
        &rrBp;
    }
}
```

```
elsif ($rcode eq "CB"
    or $rcode eq "CFp"
    or $rcode eq "Bc"
    or $rcode eq "Fcp" )
{
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("...rotation $rcode OK (= Bc = zp = Fp + Sfp + B)");
        &rrFp;
        &rrSfp;
        &rrB;
    }
}

## ----------------------------

## check empty string -->  missing rotation
elsif ( $rcode eq "" ) {
    for ( $j = 1 ; $j <= $m ; $j++ ) {
        gprint("..*rotation ,$rcode, ERROR  ? typo or missing rotation");
        ErrorMessage(",$rcode,  -- ? typo or missing rotation");
    }
}
## finally ---------------------
else {
    ## to fall this far then the rotation (char) must be undefined
    ## but before we can send these rotation code strings out in ErrorMessages
    ## we need to check that they are in the original format.
    ## ie., do not have any {;} chars etc. If they do, then we need to
    ## translate them back, ie {;} --> (,) etc

    ## we use ‘originalcode’ in the ErrorMessage because the user needs to be
    ## shown the ‘bad’ code as it was originally input by the RubikRotation{} command.

    ## check for code with { ; }  and  restore to normal syntax
    if ( $rcode =~ m/(\{|;|\}|\])/ ) {
        $rcode         = restorebrackets($rcode);
```

```
            $originalrcode = restorebrackets($originalrcode);
        }

        if ( $rcode =~ m/(\(|\)|\[|\])/ ) {
            gprint(
"..*rotation $rcode ERROR -- code not known ? missing comma or nested brackets"
            );
            ErrorMessage(
"$originalrcode  -- code not known ? missing comma or nested brackets"
            );
            ## DO NOT  --> (next LINE;) here as need to check /all/ the rotation codes in the string.
        }
        else {
            gprint(
"..*rotation $rcode ERROR -- code not known ? typo or missing comma"
            );
            ErrorMessage(
                "$originalrcode  -- code not known ? typo or missing comma");
            ## DO NOT  --> (next LINE;) here as need to check /all/ the rotation codes in the string.
        }

        #----------------------------
        next;
    }
    ;    #end of else

    next;
}    # end of sub

#====================================
```

3.3 random

```
sub random {
```

```
    print " SUB random\n";

## scramble  randomly using n rotations
## example command = RubikRotation{random,74}
## if no n given (second argument  = ""), then use default n=50
## if second argument is some string (not integer) then --> ERROR
##
## assign numbers to the minimal set of rotations to be used using a hash array list
## (perl 5 book page 68)
## ? maybe we should only use the 18 rotations mentioned in Rokicki 2013 paper?
## but here I have included all the slice (Xm) ones as well.

    ## initialise the array for the random rotations
    my @rrlist = ();

    ## (RWDN 2 Aug 2017):
    ## now check to see if TWOcube or Rubikcube being used
    ## use cubesize as the filter

    if ( $cubesize eq 'two' ) {
        ## using the TwoRotation command (from RubikTwoCube.sty)
        ## no slice rotations
        ##
        @rrlist =
          ( "U", "Up", "D", "Dp", "L", "Lp", "R", "Rp", "F", "Fp", "B", "Bp" );
    }

    else {
        ## using the RubikRotation command (from RubikRotation.sty)
        @rrlist = (
            "U", "Up", "Um", "Ump", "D", "Dp", "Dm", "Dmp",
            "L", "Lp", "Lm", "Lmp", "R", "Rp", "Rm", "Rmp",
            "F", "Fp", "Fm", "Fmp", "B", "Bp", "Bm", "Bmp"
        );
    }
```

```
    my $rrlistnumber = $#rrlist;
    print " rrlistnumber = $rrlistnumber\n";
    gprint("...random SUB: rrlistnumber (array size) =  $rrlistnumber");

    # these are numbered 0--$rrlistnumber,

## let default no of random rotations for scrambling = 50
    my $defaultn = 50;    ## RWDN (1 Aug 2017): not being used any more
    my $maxn     = 200;

    ##  grab the integer passed  from the random() command in main
    my $s = $_[0];

    if ( $s >= $maxn ) {
        $s = $maxn;
        gprint("..*WARNING: maximum n = 200");
        ErrorMessage("random: max n = 200 (n=200 was used)");
    }
    elsif ( $s == 0 ) {    ## $s = $defaultn;
        gprint("..*ERR: integer n = 0 (invalid)");
        ErrorMessage(" --- integer n = 0 (invalid)");
        next LINE;
    }

    my @rr;                ## array to hold all the random rotations
    print " randomising the available rotations\n";

    ## set the seed for the randomisation (Perl BlackBook p 235)
    srand;

## now select s numbers at random (with replacement) from range 0--listnumber+1
## Since we are using int(rand x), and  using nos from 0--lastindex number,
## then max  rand vaue = (lastindexnumber -1).99999, the  integer of which
## = (lastindexnumber -1). Therefore we need to use the range 0--(lastindexnumber+1)
## in order to randomise all posibilities on  our list.
```

```
    my $j;

    for ( $j = 1 ; $j <= $s ; $j = $j + 1 ) {
        my $p = int( rand( $rrlistnumber + 1 ) );
        print "Rotation = $p, $rrlist[$p] \n";
        ## push rotation code   $rrlist[$p] on to END of array @rr
        push( @rr, $rrlist[$p] );
    }

    ## we assume the user is starting from a solved cube (ie use the state given by user)
    gprint("...scrambling cube using $s random rotations");

    ## now send the array off to the rotation sub

    my $E;

    foreach $E (@rr) { rotation($E) }

}    ##end of sub

#=====================================
```

3.4 writestate

```
sub writestate {

    print " SUB writestate\n";

## this  writes the final state to the TeX_OUT_FILE (= rubikstateNEW.dat) will be read by latex.

    print( TeX_OUT_FILE "\%\% ...output datafile=$out_file\n" );
    print( TeX_OUT_FILE
          "\%\% ...PERL script=rubikrotation.pl version $version\n" );
    print( TeX_OUT_FILE
          "\\typeout{...writing new cube state to file $out_file}\%\n" );
```

```
    print( TeX_OUT_FILE
"\\RubikFaceUp\{$Ult[0]\}\{$Umt[0]\}\{$Urt[0]\}\{$Ulm[0]\}\{$Umm[0]\}\{$Urm[0]\}\{$Ulb[0]\}\{$Umb[0]\}\{$Urb[0]\}\%\n"
    );
    print( TeX_OUT_FILE
"\\RubikFaceDown\{$Dlt[0]\}\{$Dmt[0]\}\{$Drt[0]\}\{$Dlm[0]\}\{$Dmm[0]\}\{$Drm[0]\}\{$Dlb[0]\}\{$Dmb[0]\}\{$Drb[0]\}\%\n"
    );
    print( TeX_OUT_FILE
"\\RubikFaceLeft\{$Llt[0]\}\{$Lmt[0]\}\{$Lrt[0]\}\{$Llm[0]\}\{$Lmm[0]\}\{$Lrm[0]\}\{$Llb[0]\}\{$Lmb[0]\}\{$Lrb[0]\}\%\n"
    );
    print( TeX_OUT_FILE
"\\RubikFaceRight\{$Rlt[0]\}\{$Rmt[0]\}\{$Rrt[0]\}\{$Rlm[0]\}\{$Rmm[0]\}\{$Rrm[0]\}\{$Rlb[0]\}\{$Rmb[0]\}\{$Rrb[0]\}\%\n"
    );
    print( TeX_OUT_FILE
"\\RubikFaceFront\{$Flt[0]\}\{$Fmt[0]\}\{$Frt[0]\}\{$Flm[0]\}\{$Fmm[0]\}\{$Frm[0]\}\{$Flb[0]\}\{$Fmb[0]\}\{$Frb[0]\}\%\n"
    );
    print( TeX_OUT_FILE
"\\RubikFaceBack\{$Blt[0]\}\{$Bmt[0]\}\{$Brt[0]\}\{$Blm[0]\}\{$Bmm[0]\}\{$Brm[0]\}\{$Blb[0]\}\{$Bmb[0]\}\{$Brb[0]\}\%\n"
    );

##-----RWDN 2016---create four  new holder commands for separate strings----------

##  these four names are defined in the rubikrotation.sty file so they can be renewed etc
## SequenceInfo
## SequenceName
## SequenceShort
## SequenceLong

## ----RWDN  25 Sept 2016 ----------------------
## now remove the first and last chars of [name] to output just  NAME without [ and ]

## initialise some variables we shall need
    $numberofcharsinstring = 0;
    $nmiddlecharsinstring  = 0;

##----------SequenceName----------------------------
```

```
## the SequenceName currently includes the [..]
## need to remove the [] before senting it to LaTeX,
## so need to detect when NAME string itself is empty, eg []
## so create a variable:
    $SequenceNameNew = "";

    $numberofcharsinstring = length $SequenceName;

##   NEED to create error message if [] and  empty string etc

    if ( $numberofcharsinstring <= 2 ) { $SequenceNameNew = $SequenceName }
    else {

        $nmiddlecharsinstring = ( $numberofcharsinstring - 2 );
        ## reassign the string without first and last chars
        ### format of substr = (origstring, start possn, no of chars to use)
        $SequenceNameNew = substr( $SequenceName, 1, $nmiddlecharsinstring );
    }

    ## RWDN  24 October 2017
    ## swap char changes back before writing output string
    ## only swap the brackets
    ## (do NOT swap ; --> , as ONLY use commas /outside/ infoblocks and  between rotation sequences)
    ## swap: BBook p 138--139

    $SequenceNameNew =~ tr/\{/(/;    # swap { --> (
    $SequenceNameNew =~ tr/\}/)/;    # swap } --> )

    print( TeX_OUT_FILE
          "\\renewcommand\\SequenceName\{$SequenceNameNew\}\%\n" );

    print( TeX_OUT_FILE "\\typeout{...SequenceName = $SequenceNameNew}\%\n" );

    #-----------------

    #----------SequenceInfo----------------
```

```
## we need to  preserve any {} structures in the  info string (as used by Kociemba),
## so we have to change { } --> [ ] since otherwise they will disappear
## or cause an error when printed in LaTeX

    $SequenceInfo =~ tr/\{/\[/;    ## swap { --> [
    $SequenceInfo =~ tr/\}/\]/;    ## swap } --> ]

    print( TeX_OUT_FILE "\\renewcommand\\SequenceInfo\{$SequenceInfo\}\%\n" );

    print( TeX_OUT_FILE "\\typeout{...SequenceInfo = $SequenceInfo}\%\n" );

    #-----------------

##---------SequenceShort------------------
##  generated in  MAIN
## SequenceShort = original argument of \RubikRotation{} /without/ any infoblocks
## therefore it may contain square brackets

    print( TeX_OUT_FILE "\\renewcommand\\SequenceShort\{$SequenceShort\}\%\n" );

    print( TeX_OUT_FILE "\\typeout{...SequenceShort = $SequenceShort}\%\n" );
##-----------------------

##-----------SequenceLong-------------------
## now prepare the new LONG rotation sequence for output =(LONG sequence + NO NAME)
## BUT before outputting the string, we need to remove the terminal comma

    $numberofcharsinstring = length $SequenceLong;
    $nfrontcharsinstring   = $numberofcharsinstring - 1;
## reassign the string except the terminal comma
    $SequenceLong = substr( $SequenceLong, 0, $nfrontcharsinstring );

    #----------
    print( TeX_OUT_FILE "\\renewcommand\\SequenceLong\{$SequenceLong\}\%\n" );

    print( TeX_OUT_FILE "\\typeout{...SequenceLong = $SequenceLong}\%\n" );
```

```
##-----------------------

## now include any error messages generated
## (these are all  in an array waiting to be printed out)

    if ( $erroralert eq "YES" ) {
        ## write errors to a separate file (just for errors---we append the errrors to end of file)
        ## the error file (rubikstateERRORS.dat) was created by the TeX file
        my $ne;    #number of errors
        $ne = $#error
          ;    ## number of errors= largest index num since we started at zero

        ## do not attach error to a <checkstate> command, since we really want
        ## to see the checkstate errors (in the ERROR file) printed AFTER the 'rotation' command.
        if ( $rotationcommand eq "checkstate" ) { }
        else { print( ERROR_OUT_FILE "*ERR cmd= $rotationcommand\n" ) }

        ## last index number or array = $#arrayname (Black book p 62)
        my $k;

        for ( $k = 0 ; $k <= $ne ; $k = $k + 1 ) {

            ## restore correct brackets etc before outputting to Latex
            my $errorstring = $error[$k];
            $errorstring = restorebrackets($errorstring);

            print( TeX_OUT_FILE "\\typeout{$errorstring}\%\n" );
            print( ERROR_OUT_FILE "$errorstring\n" );

        };    # end of for
    }
    ;         # end of IF
    print " Perl output file written OK\n";

}    #end of sub
```

```
#=====================================
```

3.5 ErrorMessage

This subroutine places error messages which are generated into an array, the elements of which will be written (later) to the file `rubikstateERRORS.dat`.

```
sub ErrorMessage {

    ## writes the argument as a standard error message to out file

    my $errormess = $_[0];    ## parameter passed to sub

    ## restore correct brackets etc before outputting to Latex
    $errormess = restorebrackets($errormess);

    $erroralert = "YES";      ## set error alert flag (for use in out message)
    $error[$errornumber] = "*ERR      $errormess";
    $errornumber++;           ## increment number
}

#=====================================
```

3.6 gprint

```
sub gprint {

##  prints argument (comments) to screen and also to TeX_OUT_FILE.
## The typeout commands will find its way into the log file when read by latex
## Important to include trailing % for messages written to the TeX_OUT_FILE
##    to stop extra <spaces> being seen by TeX.
```

```
    my $gmess = $_[0];
    print "$gmess\n";
    print( TeX_OUT_FILE "\\typeout{$gmess}\%\n" );
}

#====================================
```

3.7 checkstate

```
sub checkstate {

    print " SUB checkstate\n";

## only a simple check -- to see if wrong no of colours being used etc
## uses the cubie colours as used by rubikcube package= ROYGBWX

    gprint("...checking state of cube");

    my @cubies = (
        $Ult[0], $Umt[0], $Urt[0], $Ulm[0], $Umm[0], $Urm[0], $Ulb[0],
        $Umb[0], $Urb[0], $Dlt[0], $Dmt[0], $Drt[0], $Dlm[0], $Dmm[0],
        $Drm[0], $Dlb[0], $Dmb[0], $Drb[0], $Llt[0], $Lmt[0], $Lrt[0],
        $Llm[0], $Lmm[0], $Lrm[0], $Llb[0], $Lmb[0], $Lrb[0], $Rlt[0],
        $Rmt[0], $Rrt[0], $Rlm[0], $Rmm[0], $Rrm[0], $Rlb[0], $Rmb[0],
        $Rrb[0], $Flt[0], $Fmt[0], $Frt[0], $Flm[0], $Fmm[0], $Frm[0],
        $Flb[0], $Fmb[0], $Frb[0], $Blt[0], $Bmt[0], $Brt[0], $Blm[0],
        $Bmm[0], $Brm[0], $Blb[0], $Bmb[0], $Brb[0]
    );

    my $R = 0, my $O = 0, my $Y = 0, my $G = 0, my $B = 0, my $W = 0, my $X = 0;

    my $cubiecolour = "";

    foreach $cubiecolour (@cubies) {
        if    ( $cubiecolour eq R ) { $R = $R + 1 }
```

```
        elsif ( $cubiecolour eq O ) { $O = $O + 1 }
        elsif ( $cubiecolour eq Y ) { $Y = $Y + 1 }
        elsif ( $cubiecolour eq G ) { $G = $G + 1 }
        elsif ( $cubiecolour eq B ) { $B = $B + 1 }
        elsif ( $cubiecolour eq W ) { $W = $W + 1 }
        elsif ( $cubiecolour eq X ) { $X = $X + 1 }
        else {
            gprint("..*cubie-colour counting ERROR");
        }
    }

    my $cubiesum = 0;
    $cubiesum = $R + $O + $Y + $G + $B + $W + $X;
    gprint(
"...cubiesum = $cubiesum (Red=$R, Or=$O, Ye=$Y, Gr=$G, Bl=$B, Wh=$W, X=$X)"
    );

    # only generate ErrorMessages if n>9 (as may be using a Grey cube)
    if ( $cubiesum != 54 ) {
        ErrorMessage("cubiesum not = 54");
        gprint("..*cubiesum not = 54");
    }

    if ( $R > 9 ) {
        ErrorMessage("red cubies > 9 (=$R)");
        gprint("..*red cubies > 9 (=$R)");
    }

    if ( $O > 9 ) {
        ErrorMessage("orange cubies > 9 (=$O)");
        gprint("..*orange cubies > 9 (=$O)");
    }

    if ( $Y > 9 ) {
        ErrorMessage("yellow cubies > 9 (=$Y)");
        gprint("..*yellow cubies > 9 (=$Y)");
```

```
        #  next LINE
    }

    if ( $G > 9 ) {
        ErrorMessage("green cubies > 9 (=$G)");
        gprint("..*green cubies > 9 (=$G)");
    }

    if ( $B > 9 ) {
        ErrorMessage("blue cubies > 9 (=$B)");
        gprint("..*blue cubies > 9 (=$B)");
    }

    if ( $W > 9 ) {
        ErrorMessage("white cubies > 9 (=$W)");
        gprint("..*white cubies > 9 (=$W)");
    }

    if ( $X == 54 ) {
        ErrorMessage("no colours allocated  (X=54)");
        gprint("..*no colours allocated  (X=54)");
    }

    print " done\n\n";
}
```

3.8 rr-overview of rotation subs

```
## The following 9 subs (90 degree rotation transformations) are used
##   to generate all the rotations used in the 'rotation sub' (see above).
## Each of these is a permutation for both colours and numbers of the cubie facelets.
## The following 9 subroutines are named as follows:
##   (about X-axis) rrR, rrSr, rrLp
```

```
##   (about Y-axis) rrU, rrSu, rrDp
##   (about Z-axis) rrF, rrSf, rrBp
##   see the rubikcube package documentation for full details regarding rotation notation and commands.
##
## METHOD & NOTATION
## Each sub (below) starts by making an array[0] for the cubie colour
##   and an array[1] for  the cubie number.
## Each  of the face rotations (rrR, rrLp, rrU, rrDp, rrF, rrBp) is involved with
##   two pairs of connected but different permutations/transformations as follows:
##   (a) one pair for the 12 'Side' cubies (arrays = @Xs0 (for Side colours), @Xs1 (for Side numbers)), and
##   (b) one pair for the 9 'Face' cubies (arrays = @Xf0 (for Face colours), @Xf1 (for Face numbers)).
## Each of the center slice rotations (rrSr, rrSu, rrSf)  is involved with just one pair of
##   permutations for the 12 Side cubies (arrays = @Xs0 (for Side colours), @Xs1 (for Side numbers)).
## We document only the Side and Face of the first sub (rrR) in detail, since the other subs are
##   of similar form.
##=========================================
```

3.9 rrR

See subsections **rr–overview** (above) for details of notation.

```
sub rrR {

##  the RIGHT (slice + face) transform
## R = RIGHT, s = side; 0=colour, 1= number
## make the clockwise rotation permutation
## In this permutation the Front-right-bottom (Frb) (side)facelet rotates to
##    the new position of Up-right-bottom (Urb) (side)facelet.
##-----------SIDE-------
## 12 side cubie facelets in  arrays @Rs0 (colours) and @Rs1 (numbers)
## these are the initial positions

    @Rs0 = (
        $Frb[0], $Frm[0], $Frt[0], $Urb[0], $Urm[0], $Urt[0],
```

```
        $Blt[0], $Blm[0], $Blb[0], $Drb[0], $Drm[0], $Drt[0]
    );

    @Rs1 = (
        $Frb[1], $Frm[1], $Frt[1], $Urb[1], $Urm[1], $Urt[1],
        $Blt[1], $Blm[1], $Blb[1], $Drb[1], $Drm[1], $Drt[1]
    );

## now we reallocate the initial array elements to the new
##    post (90 degree clockwise) rotation position.
## Cube is viewed from FRONT.
## Positions of side facelets of Right slice are numbered 0-11 in clockwise direction,
##   (as seen from Right face) starting with Up-right-bottom facelet.
## First line example:
## variable $Urb[0] (Upface-right-bottom colour) <-- colour of first element in @Rs0 (=Frb[0])
## variable $Urb[1] (Upface-right-bottom number) <-- number of first element in @Rs1 (=Frb[1])

    $Urb[0] = $Rs0[0];
    $Urb[1] = $Rs1[0];
    $Urm[0] = $Rs0[1];
    $Urm[1] = $Rs1[1];
    $Urt[0] = $Rs0[2];
    $Urt[1] = $Rs1[2];

    $Blt[0] = $Rs0[3];
    $Blt[1] = $Rs1[3];
    $Blm[0] = $Rs0[4];
    $Blm[1] = $Rs1[4];
    $Blb[0] = $Rs0[5];
    $Blb[1] = $Rs1[5];

    $Drb[0] = $Rs0[6];
    $Drb[1] = $Rs1[6];
    $Drm[0] = $Rs0[7];
    $Drm[1] = $Rs1[7];
    $Drt[0] = $Rs0[8];
```

```
    $Drt[1] = $Rs1[8];

    $Frb[0] = $Rs0[9];
    $Frb[1] = $Rs1[9];
    $Frm[0] = $Rs0[10];
    $Frm[1] = $Rs1[10];
    $Frt[0] = $Rs0[11];
    $Frt[1] = $Rs1[11];

##-------------Right FACE---------------------
## RIGHT FACE (9 cubies in each array)
## (numbered in rows: 1,2,3/4,5,6/7,8,9 from top left(1) to bottom right(9))
## R=Right, f = face; 0=colour, 1= number
## do the Rface (90 degree) rotation transform
## here  the Right-left-bottom (Rlb) facelet rotates to the possn of Right-left-top (Rlt)
## we start with two arrays (one for colours @Rf0, one for numbers @Rf1)  with 9 elements each.

    @Rf0 = (
        $Rlb[0], $Rlm[0], $Rlt[0], $Rmb[0], $Rmm[0],
        $Rmt[0], $Rrb[0], $Rrm[0], $Rrt[0]
    );
    @Rf1 = (
        $Rlb[1], $Rlm[1], $Rlt[1], $Rmb[1], $Rmm[1],
        $Rmt[1], $Rrb[1], $Rrm[1], $Rrt[1]
    );

## now we reallocate the  array elements to the new
## post (90 degree clockwise) rotation  facelet position.
## Right face is viewed from RIGHT.
## First line example:
## variable $Rlt[0] (=Right-left-top colour) <-- colour of first element in @Rf0 (=Rlb[0])
## variable $Rlt[1] (=Right-left-top number) <-- number of first element in @Rf1 (=Rlb[1])

    $Rlt[0] = $Rf0[0];
    $Rlt[1] = $Rf1[0];
    $Rmt[0] = $Rf0[1];
```

```
    $Rmt[1] = $Rf1[1];
    $Rrt[0] = $Rf0[2];
    $Rrt[1] = $Rf1[2];

    $Rlm[0] = $Rf0[3];
    $Rlm[1] = $Rf1[3];
    $Rmm[0] = $Rf0[4];
    $Rmm[1] = $Rf1[4];
    $Rrm[0] = $Rf0[5];
    $Rrm[1] = $Rf1[5];

    $Rlb[0] = $Rf0[6];
    $Rlb[1] = $Rf1[6];
    $Rmb[0] = $Rf0[7];
    $Rmb[1] = $Rf1[7];
    $Rrb[0] = $Rf0[8];
    $Rrb[1] = $Rf1[8];

}

#=====================================
```

3.10 rrSr

See subsections **rr–overview** and **rrR** (above) for details of notation.

```
sub rrSr {

## Sr = RIGHT middle SLICE rotation (only 12 side facelets)
## modified from rrR (change the U,D,F, r --> m and Back Bl-->Bm; Rs--> ?Srs)
## change only the slice
## s = side; 0=colour, 1= number
## make the post rotation permutation

    @SRs0 = (
```

```
    $Fmb[0], $Fmm[0], $Fmt[0], $Umb[0], $Umm[0], $Umt[0],
    $Bmt[0], $Bmm[0], $Bmb[0], $Dmb[0], $Dmm[0], $Dmt[0]
);

@SRs1 = (
    $Fmb[1], $Fmm[1], $Fmt[1], $Umb[1], $Umm[1], $Umt[1],
    $Bmt[1], $Bmm[1], $Bmb[1], $Dmb[1], $Dmm[1], $Dmt[1]
);

$Umb[0] = $SRs0[0];
$Umb[1] = $SRs1[0];
$Umm[0] = $SRs0[1];
$Umm[1] = $SRs1[1];
$Umt[0] = $SRs0[2];
$Umt[1] = $SRs1[2];

$Bmt[0] = $SRs0[3];
$Bmt[1] = $SRs1[3];
$Bmm[0] = $SRs0[4];
$Bmm[1] = $SRs1[4];
$Bmb[0] = $SRs0[5];
$Bmb[1] = $SRs1[5];

$Dmb[0] = $SRs0[6];
$Dmb[1] = $SRs1[6];
$Dmm[0] = $SRs0[7];
$Dmm[1] = $SRs1[7];
$Dmt[0] = $SRs0[8];
$Dmt[1] = $SRs1[8];

$Fmb[0] = $SRs0[9];
$Fmb[1] = $SRs1[9];
$Fmm[0] = $SRs0[10];
$Fmm[1] = $SRs1[10];
$Fmt[0] = $SRs0[11];
$Fmt[1] = $SRs1[11];
```

```
}

#====================================
```

3.11 rrLp

See subsections **rr–overview** and **rrR** (above) for details of notation.

```
sub rrLp {

## LEFT slice (side + face) anticlockwise rotation
## s = side; 0=colour, 1= number
##-------------side-----------

    @LPs0 = (
        $Flb[0], $Flm[0], $Flt[0], $Ulb[0], $Ulm[0], $Ult[0],
        $Brt[0], $Brm[0], $Brb[0], $Dlb[0], $Dlm[0], $Dlt[0]
    );

    @LPs1 = (
        $Flb[1], $Flm[1], $Flt[1], $Ulb[1], $Ulm[1], $Ult[1],
        $Brt[1], $Brm[1], $Brb[1], $Dlb[1], $Dlm[1], $Dlt[1]
    );

    $Ulb[0] = $LPs0[0];
    $Ulb[1] = $LPs1[0];
    $Ulm[0] = $LPs0[1];
    $Ulm[1] = $LPs1[1];
    $Ult[0] = $LPs0[2];
    $Ult[1] = $LPs1[2];

    $Brt[0] = $LPs0[3];
    $Brt[1] = $LPs1[3];
    $Brm[0] = $LPs0[4];
```

```
    $Brm[1] = $LPs1[4];
    $Brb[0] = $LPs0[5];
    $Brb[1] = $LPs1[5];

    $Dlb[0] = $LPs0[6];
    $Dlb[1] = $LPs1[6];
    $Dlm[0] = $LPs0[7];
    $Dlm[1] = $LPs1[7];
    $Dlt[0] = $LPs0[8];
    $Dlt[1] = $LPs1[8];

    $Flb[0] = $LPs0[9];
    $Flb[1] = $LPs1[9];
    $Flm[0] = $LPs0[10];
    $Flm[1] = $LPs1[10];
    $Flt[0] = $LPs0[11];
    $Flt[1] = $LPs1[11];

##---------------Left FACE-------------
## do the LEFT face transform (in rows: 1,2,3//4,5,6//7,8,9)
## f = face; 0=colour, 1= number
## NOTES: not same as for R

    @LPf0 = (
        $Lrt[0], $Lrm[0], $Lrb[0], $Lmt[0], $Lmm[0],
        $Lmb[0], $Llt[0], $Llm[0], $Llb[0]
    );
    @LPf1 = (
        $Lrt[1], $Lrm[1], $Lrb[1], $Lmt[1], $Lmm[1],
        $Lmb[1], $Llt[1], $Llm[1], $Llb[1]
    );

    $Llt[0] = $LPf0[0];
    $Llt[1] = $LPf1[0];
    $Lmt[0] = $LPf0[1];
    $Lmt[1] = $LPf1[1];
```

```
    $Lrt[0] = $LPf0[2];
    $Lrt[1] = $LPf1[2];

    $Llm[0] = $LPf0[3];
    $Llm[1] = $LPf1[3];
    $Lmm[0] = $LPf0[4];
    $Lmm[1] = $LPf1[4];
    $Lrm[0] = $LPf0[5];
    $Lrm[1] = $LPf1[5];

    $Llb[0] = $LPf0[6];
    $Llb[1] = $LPf1[6];
    $Lmb[0] = $LPf0[7];
    $Lmb[1] = $LPf1[7];
    $Lrb[0] = $LPf0[8];
    $Lrb[1] = $LPf1[8];

}

#=====================================
```

3.12 rrU

See subsections **rr–overview** and **rrR** (above) for details of notation.

```
sub rrU {

## UP slice (side + face)
## do the Uside transform
## s = side; 0=colour, 1= number
## ----------SIDE--------------

    @Us0 = (
        $Lrt[0], $Lmt[0], $Llt[0], $Brt[0], $Bmt[0], $Blt[0],
        $Rrt[0], $Rmt[0], $Rlt[0], $Frt[0], $Fmt[0], $Flt[0]
```

```
    );

    @Us1 = (
        $Lrt[1], $Lmt[1], $Llt[1], $Brt[1], $Bmt[1], $Blt[1],
        $Rrt[1], $Rmt[1], $Rlt[1], $Frt[1], $Fmt[1], $Flt[1]
    );

    $Brt[0] = $Us0[0];
    $Brt[1] = $Us1[0];
    $Bmt[0] = $Us0[1];
    $Bmt[1] = $Us1[1];
    $Blt[0] = $Us0[2];
    $Blt[1] = $Us1[2];

    $Rrt[0] = $Us0[3];
    $Rrt[1] = $Us1[3];
    $Rmt[0] = $Us0[4];
    $Rmt[1] = $Us1[4];
    $Rlt[0] = $Us0[5];
    $Rlt[1] = $Us1[5];

    $Frt[0] = $Us0[6];
    $Frt[1] = $Us1[6];
    $Fmt[0] = $Us0[7];
    $Fmt[1] = $Us1[7];
    $Flt[0] = $Us0[8];
    $Flt[1] = $Us1[8];

    $Lrt[0] = $Us0[9];
    $Lrt[1] = $Us1[9];
    $Lmt[0] = $Us0[10];
    $Lmt[1] = $Us1[10];
    $Llt[0] = $Us0[11];
    $Llt[1] = $Us1[11];

##-------------Up FACE-------------------
```

```
## do the Rface transform (in rows: 1,2,3//4,5,6//7,8,9)
## f = face; 0=colour, 1= number

    @Uf0 = (
        $Ulb[0], $Ulm[0], $Ult[0], $Umb[0], $Umm[0],
        $Umt[0], $Urb[0], $Urm[0], $Urt[0]
    );
    @Uf1 = (
        $Ulb[1], $Ulm[1], $Ult[1], $Umb[1], $Umm[1],
        $Umt[1], $Urb[1], $Urm[1], $Urt[1]
    );

    $Ult[0] = $Uf0[0];
    $Ult[1] = $Uf1[0];
    $Umt[0] = $Uf0[1];
    $Umt[1] = $Uf1[1];
    $Urt[0] = $Uf0[2];
    $Urt[1] = $Uf1[2];

    $Ulm[0] = $Uf0[3];
    $Ulm[1] = $Uf1[3];
    $Umm[0] = $Uf0[4];
    $Umm[1] = $Uf1[4];
    $Urm[0] = $Uf0[5];
    $Urm[1] = $Uf1[5];

    $Ulb[0] = $Uf0[6];
    $Ulb[1] = $Uf1[6];
    $Umb[0] = $Uf0[7];
    $Umb[1] = $Uf1[7];
    $Urb[0] = $Uf0[8];
    $Urb[1] = $Uf1[8];

}

#=======================================
```

3.13 rrSu

See subsections **rr–overview** and **rrR** (above) for details of notation.

```
sub rrSu {

## middle slice rotation (side only 12 facelets)
## s = side; 0=colour, 1= number
## make the post rotation permutation
##-----------SIDE------------------

    @SUs0 = (
        $Lrm[0], $Lmm[0], $Llm[0], $Brm[0], $Bmm[0], $Blm[0],
        $Rrm[0], $Rmm[0], $Rlm[0], $Frm[0], $Fmm[0], $Flm[0]
    );

    @SUs1 = (
        $Lrm[1], $Lmm[1], $Llm[1], $Brm[1], $Bmm[1], $Blm[1],
        $Rrm[1], $Rmm[1], $Rlm[1], $Frm[1], $Fmm[1], $Flm[1]
    );

    $Brm[0] = $SUs0[0];
    $Brm[1] = $SUs1[0];
    $Bmm[0] = $SUs0[1];
    $Bmm[1] = $SUs1[1];
    $Blm[0] = $SUs0[2];
    $Blm[1] = $SUs1[2];

    $Rrm[0] = $SUs0[3];
    $Rrm[1] = $SUs1[3];
    $Rmm[0] = $SUs0[4];
    $Rmm[1] = $SUs1[4];
    $Rlm[0] = $SUs0[5];
    $Rlm[1] = $SUs1[5];
```

```
    $Frm[0] = $SUs0[6];
    $Frm[1] = $SUs1[6];
    $Fmm[0] = $SUs0[7];
    $Fmm[1] = $SUs1[7];
    $Flm[0] = $SUs0[8];
    $Flm[1] = $SUs1[8];

    $Lrm[0] = $SUs0[9];
    $Lrm[1] = $SUs1[9];
    $Lmm[0] = $SUs0[10];
    $Lmm[1] = $SUs1[10];
    $Llm[0] = $SUs0[11];
    $Llm[1] = $SUs1[11];

}

#=======================================
```

3.14 rrDp

See subsections **rr–overview** and **rrR** (above) for details of notation.

```
sub rrDp {

## Down Face anticlockwise rotation (side and face)
## s = side; 0=colour, 1= number
## make the post rotation permutation
##--------------SIDE----------------

    @DPs0 = (
        $Lrb[0], $Lmb[0], $Llb[0], $Brb[0], $Bmb[0], $Blb[0],
        $Rrb[0], $Rmb[0], $Rlb[0], $Frb[0], $Fmb[0], $Flb[0]
    );
```

```
    @DPs1 = (
        $Lrb[1], $Lmb[1], $Llb[1], $Brb[1], $Bmb[1], $Blb[1],
        $Rrb[1], $Rmb[1], $Rlb[1], $Frb[1], $Fmb[1], $Flb[1]
    );

    $Brb[0] = $DPs0[0];
    $Brb[1] = $DPs1[0];
    $Bmb[0] = $DPs0[1];
    $Bmb[1] = $DPs1[1];
    $Blb[0] = $DPs0[2];
    $Blb[1] = $DPs1[2];

    $Rrb[0] = $DPs0[3];
    $Rrb[1] = $DPs1[3];
    $Rmb[0] = $DPs0[4];
    $Rmb[1] = $DPs1[4];
    $Rlb[0] = $DPs0[5];
    $Rlb[1] = $DPs1[5];

    $Frb[0] = $DPs0[6];
    $Frb[1] = $DPs1[6];
    $Fmb[0] = $DPs0[7];
    $Fmb[1] = $DPs1[7];
    $Flb[0] = $DPs0[8];
    $Flb[1] = $DPs1[8];

    $Lrb[0] = $DPs0[9];
    $Lrb[1] = $DPs1[9];
    $Lmb[0] = $DPs0[10];
    $Lmb[1] = $DPs1[10];
    $Llb[0] = $DPs0[11];
    $Llb[1] = $DPs1[11];

##---------------Down FACE-------------------
## f = face; 0=colour, 1= number
```

```
    @DPf0 = (
        $Dlt[0], $Dlm[0], $Dlb[0], $Dmt[0], $Dmm[0],
        $Dmb[0], $Drt[0], $Drm[0], $Drb[0]
    );
    @DPf1 = (
        $Dlt[1], $Dlm[1], $Dlb[1], $Dmt[1], $Dmm[1],
        $Dmb[1], $Drt[1], $Drm[1], $Drb[1]
    );

    $Dlb[0] = $DPf0[0];
    $Dlb[1] = $DPf1[0];
    $Dmb[0] = $DPf0[1];
    $Dmb[1] = $DPf1[1];
    $Drb[0] = $DPf0[2];
    $Drb[1] = $DPf1[2];

    $Dlm[0] = $DPf0[3];
    $Dlm[1] = $DPf1[3];
    $Dmm[0] = $DPf0[4];
    $Dmm[1] = $DPf1[4];
    $Drm[0] = $DPf0[5];
    $Drm[1] = $DPf1[5];

    $Dlt[0] = $DPf0[6];
    $Dlt[1] = $DPf1[6];
    $Dmt[0] = $DPf0[7];
    $Dmt[1] = $DPf1[7];
    $Drt[0] = $DPf0[8];
    $Drt[1] = $DPf1[8];

}

#=======================================
```

3.15 rrF

See subsections **rr–overview** and **rrR** (above) for details of notation.

```
sub rrF {

## do the Fside transform (side and face)
## s = side; 0=colour, 1= number
## -----------SIDE-----------------

    @Fs0 = (
        $Lrb[0], $Lrm[0], $Lrt[0], $Ulb[0], $Umb[0], $Urb[0],
        $Rlt[0], $Rlm[0], $Rlb[0], $Drt[0], $Dmt[0], $Dlt[0]
    );

    @Fs1 = (
        $Lrb[1], $Lrm[1], $Lrt[1], $Ulb[1], $Umb[1], $Urb[1],
        $Rlt[1], $Rlm[1], $Rlb[1], $Drt[1], $Dmt[1], $Dlt[1]
    );

    $Ulb[0] = $Fs0[0];
    $Ulb[1] = $Fs1[0];
    $Umb[0] = $Fs0[1];
    $Umb[1] = $Fs1[1];
    $Urb[0] = $Fs0[2];
    $Urb[1] = $Fs1[2];

    $Rlt[0] = $Fs0[3];
    $Rlt[1] = $Fs1[3];
    $Rlm[0] = $Fs0[4];
    $Rlm[1] = $Fs1[4];
    $Rlb[0] = $Fs0[5];
    $Rlb[1] = $Fs1[5];

    $Drt[0] = $Fs0[6];
    $Drt[1] = $Fs1[6];
    $Dmt[0] = $Fs0[7];
```

```
    $Dmt[1] = $Fs1[7];
    $Dlt[0] = $Fs0[8];
    $Dlt[1] = $Fs1[8];

    $Lrb[0] = $Fs0[9];
    $Lrb[1] = $Fs1[9];
    $Lrm[0] = $Fs0[10];
    $Lrm[1] = $Fs1[10];
    $Lrt[0] = $Fs0[11];
    $Lrt[1] = $Fs1[11];

## -------Front FACE--------------------
## f = face; 0=colour, 1= number

    @Lf0 = (
        $Flb[0], $Flm[0], $Flt[0], $Fmb[0], $Fmm[0],
        $Fmt[0], $Frb[0], $Frm[0], $Frt[0]
    );
    @Lf1 = (
        $Flb[1], $Flm[1], $Flt[1], $Fmb[1], $Fmm[1],
        $Fmt[1], $Frb[1], $Frm[1], $Frt[1]
    );

    $Flt[0] = $Lf0[0];
    $Flt[1] = $Lf1[0];
    $Fmt[0] = $Lf0[1];
    $Fmt[1] = $Lf1[1];
    $Frt[0] = $Lf0[2];
    $Frt[1] = $Lf1[2];

    $Flm[0] = $Lf0[3];
    $Flm[1] = $Lf1[3];
    $Fmm[0] = $Lf0[4];
    $Fmm[1] = $Lf1[4];
    $Frm[0] = $Lf0[5];
    $Frm[1] = $Lf1[5];
```

```
    $Flb[0] = $Lf0[6];
    $Flb[1] = $Lf1[6];
    $Fmb[0] = $Lf0[7];
    $Fmb[1] = $Lf1[7];
    $Frb[0] = $Lf0[8];
    $Frb[1] = $Lf1[8];

}

#=====================================
```

3.16 rrSf

See subsections **rr–overview** and **rrR** (above) for details of notation.

```
sub rrSf {

## do the FRONT middle slice  Fm  transform (side only)
## s = side; 0=colour, 1= number
##----------SIDE---------------

    @SFs0 = (
        $Lmb[0], $Lmm[0], $Lmt[0], $Ulm[0], $Umm[0], $Urm[0],
        $Rmt[0], $Rmm[0], $Rmb[0], $Drm[0], $Dmm[0], $Dlm[0]
    );

    @SFs1 = (
        $Lmb[1], $Lmm[1], $Lmt[1], $Ulm[1], $Umm[1], $Urm[1],
        $Rmt[1], $Rmm[1], $Rmb[1], $Drm[1], $Dmm[1], $Dlm[1]
    );

    $Ulm[0] = $SFs0[0];
    $Ulm[1] = $SFs1[0];
    $Umm[0] = $SFs0[1];
```

```
    $Umm[1] = $SFs1[1];
    $Urm[0] = $SFs0[2];
    $Urm[1] = $SFs1[2];

    $Rmt[0] = $SFs0[3];
    $Rmt[1] = $SFs1[3];
    $Rmm[0] = $SFs0[4];
    $Rmm[1] = $SFs1[4];
    $Rmb[0] = $SFs0[5];
    $Rmb[1] = $SFs1[5];

    $Drm[0] = $SFs0[6];
    $Drm[1] = $SFs1[6];
    $Dmm[0] = $SFs0[7];
    $Dmm[1] = $SFs1[7];
    $Dlm[0] = $SFs0[8];
    $Dlm[1] = $SFs1[8];

    $Lmb[0] = $SFs0[9];
    $Lmb[1] = $SFs1[9];
    $Lmm[0] = $SFs0[10];
    $Lmm[1] = $SFs1[10];
    $Lmt[0] = $SFs0[11];
    $Lmt[1] = $SFs1[11];
}

#=======================================
```

3.17 rrBp

See subsections **rr–overview** and **rrR** (above) for details of notation.

```
sub rrBp {

## Back rotation anticlockwise (side + face)
```

```
## do the Bp side transform
## s = side; 0=colour, 1= number
## --------------Side-----------------

    @BPs0 = (
        $Llb[0], $Llm[0], $Llt[0], $Ult[0], $Umt[0], $Urt[0],
        $Rrt[0], $Rrm[0], $Rrb[0], $Drb[0], $Dmb[0], $Dlb[0]
    );

    @BPs1 = (
        $Llb[1], $Llm[1], $Llt[1], $Ult[1], $Umt[1], $Urt[1],
        $Rrt[1], $Rrm[1], $Rrb[1], $Drb[1], $Dmb[1], $Dlb[1]
    );

    $Ult[0] = $BPs0[0];
    $Ult[1] = $BPs1[0];
    $Umt[0] = $BPs0[1];
    $Umt[1] = $BPs1[1];
    $Urt[0] = $BPs0[2];
    $Urt[1] = $BPs1[2];

    $Rrt[0] = $BPs0[3];
    $Rrt[1] = $BPs1[3];
    $Rrm[0] = $BPs0[4];
    $Rrm[1] = $BPs1[4];
    $Rrb[0] = $BPs0[5];
    $Rrb[1] = $BPs1[5];

    $Drb[0] = $BPs0[6];
    $Drb[1] = $BPs1[6];
    $Dmb[0] = $BPs0[7];
    $Dmb[1] = $BPs1[7];
    $Dlb[0] = $BPs0[8];
    $Dlb[1] = $BPs1[8];

    $Llb[0] = $BPs0[9];
```

```
    $Llb[1] = $BPs1[9];
    $Llm[0] = $BPs0[10];
    $Llm[1] = $BPs1[10];
    $Llt[0] = $BPs0[11];
    $Llt[1] = $BPs1[11];

##-----------------Back FACE-------------
## do the B face transform (in rows: 1,2,3/4,5,6/7,8,9)
## f = face; 0=colour, 1= number

    @BPf0 = (
        $Brb[0], $Brm[0], $Brt[0], $Bmb[0], $Bmm[0],
        $Bmt[0], $Blb[0], $Blm[0], $Blt[0]
    );

    @BPf1 = (
        $Brb[1], $Brm[1], $Brt[1], $Bmb[1], $Bmm[1],
        $Bmt[1], $Blb[1], $Blm[1], $Blt[1]
    );

    $Brt[0] = $BPf0[0];
    $Brt[1] = $BPf1[0];
    $Bmt[0] = $BPf0[1];
    $Bmt[1] = $BPf1[1];
    $Blt[0] = $BPf0[2];
    $Blt[1] = $BPf1[2];

    $Brm[0] = $BPf0[3];
    $Brm[1] = $BPf1[3];
    $Bmm[0] = $BPf0[4];
    $Bmm[1] = $BPf1[4];
    $Blm[0] = $BPf0[5];
    $Blm[1] = $BPf1[5];

    $Brb[0] = $BPf0[6];
    $Brb[1] = $BPf1[6];
```

```
    $Bmb[0] = $BPf0[7];
    $Bmb[1] = $BPf1[7];
    $Blb[0] = $BPf0[8];
    $Blb[1] = $BPf1[8];
}
```

3.18 rr-overview of derivative subs

```
##  Note that we have already defined (as rotation SUBs above) just 9 primary rotation
##  transforms, namely:
##  (x axis): rrR, rrSr, rrLp,
##  (y axis): rrU, rrSu, rrDp,
##  (z axis): rrF, rrSf, rrBp,
##  and since all remaining possible rotations are simply combinations of these 9
##  we now define all the other rotation subs in terms of these 9 primary rotations.
##  Do NOT use multiples here: write each rotation separately
##  Note that Sr, Su, Sf are  middle slice rotations (= Rm, Um, Fm respectively).
##=========================================
```

3.19 rrRp

```
sub rrRp { &rrR; &rrR; &rrR };    # (=rrR3)
```

3.20 rrRw

```
sub rrRw  { &rrR;  &rrSr };                              # (= rrR + rrSr)
```

3.21 rrRwp

```
sub rrRwp { &rrR;  &rrR; &rrR; &rrSr; &rrSr; &rrSr };    # (= rrRp + rrSrp)
```

3.22 rrRs

```
sub rrRs  { &rrR;  &rrLp }
```

3.23 rrRsp

```
sub rrRsp { &rrRp; &rrL }
```

3.24 rrRa

```
sub rrRa  { &rrR;  &rrL }
```

3.25 rrRap

```
sub rrRap { &rrRp; &rrLp }

## ---------------------
```

3.26 rrL

```
sub rrL { &rrLp; &rrLp; &rrLp };                    # (= rrLp3)
```

3.27 rrLw

```
sub rrLw { &rrLp; &rrLp; &rrLp; &rrSrp };           # (=rrLp3 + rrSrp)
```

3.28 rrLwp

```
sub rrLwp { &rrLp; &rrSr }
```

3.29 rrLs

```
sub rrLs  { &rrL;  &rrRp }
```

3.30 rrLsp

```
sub rrLsp { &rrLp; &rrR }
```

3.31 rrLa

```
sub rrLa  { &rrL;  &rrR }
```

3.32 rrLap

```
sub rrLap { &rrLp; &rrRp }

## ----derivative subs from U ----
```

3.33 rrUp

```
sub rrUp { &rrU; &rrU; &rrU };                          # (=rrU3)
```

3.34 rrUw

```
sub rrUw  { &rrU;  &rrSu };                             #
```

3.35 rrUwp

```
sub rrUwp { &rrUp; &rrSup }
```

3.36 rrUs

```
sub rrUs  { &rrU;  &rrDp }
```

3.37 rrUsp

```
sub rrUsp { &rrUp; &rrD }
```

3.38 rrUa

```
sub rrUa  { &rrU;  &rrD }
```

3.39 rrUap

```
sub rrUap { &rrUp; &rrDp }

## ---------------------
```

3.40 rrD

```
sub rrD { &rrDp; &rrDp; &rrDp };                    # (= rrDp3)
```

3.41 rrDw

```
sub rrDw { &rrDp; &rrDp; &rrDp; &rrSup };           # (=rrDp3 + rrSup)
```

3.42 rrDwp

```
sub rrDwp { &rrDp; &rrSu }
```

3.43 rrDs

```
sub rrDs  { &rrD;  &rrUp }
```

3.44 rrDsp

```
sub rrDsp { &rrDp; &rrU }
```

3.45 rrDa

```
sub rrDa  { &rrD;  &rrU }
```

3.46 rrDap

```
sub rrDap { &rrDp; &rrUp }

##  ----derivative subs from F ----
```

3.47 rrFw

```
sub rrFw  { &rrF;  &rrSf };                          # (= rrF + rrSf)
```

3.48 rrFp

```
sub rrFp  { &rrF;  &rrF; &rrF };                     # (=rrF3)
```

3.49 rrFwp

```
sub rrFwp { &rrF;  &rrF; &rrF; &rrSf; &rrSf; &rrSf };    # (= rrF3 + rrSf3)
```

3.50 rrFs

```
sub rrFs  { &rrF;  &rrBp }
```

3.51 rrFsp

```
sub rrFsp { &rrFp; &rrB }
```

3.52 rrFa

```
sub rrFa  { &rrF;  &rrB }
```

3.53 rrFap

```
sub rrFap { &rrFp; &rrBp }

## --------------------
```

3.54 rrB

```
sub rrB { &rrBp; &rrBp; &rrBp };                     # (= rrBp3)
```

3.55 rrBw

```
sub rrBw { &rrBp; &rrBp; &rrBp; &rrSfp };            # (=rrBp3 + rrSfp)
```

3.56 rrBwp

```
sub rrBwp { &rrBp; &rrSf }
```

3.57 rrBs

```
sub rrBs  { &rrB;  &rrFp }
```

3.58 rrBsp

```
sub rrBsp { &rrBp; &rrF }
```

3.59 rrBa

```
sub rrBa  { &rrB;  &rrF }
```

3.60 rrBap

```
sub rrBap { &rrBp; &rrFp }

## ----bring all the S versions together ----
```

3.61 rrSup

```
sub rrSup { &rrSu; &rrSu; &rrSu };                    # (=rrSu3)
```

3.62 rrSd

```
sub rrSd  { &rrSup };                                 # (=rrSup)
```

3.63 rrSdp

```
sub rrSdp { &rrSu };                                  # (=rrSu)
```

3.64 rrSl

```
sub rrSl  { &rrSrp };                                 # (=rrSrp)
```

3.65 rrSlp

```
sub rrSlp { &rrSr };                                    # (=rrSr)
```

3.66 rrSrp

```
sub rrSrp { &rrSr; &rrSr; &rrSr };                      # (=rrSr3)
```

3.67 rrSfp

```
sub rrSfp { &rrSf; &rrSf; &rrSf };                      # (=rrSf3)
```

3.68 rrSb

```
sub rrSb  { &rrSfp };                                   # (=rrSfp)
```

3.69 rrSbp

```
sub rrSbp { &rrSf };                                    # (=rrSf)

#=====================================
```

3.70 rubikmod

This subroutine processes a rotation code (e.g., `Dp257`) and splits it into the terminal numeric rotation number (→ `$rotnumber`) and the (remaining) front part (→ `$rotcode`). It then determines the value modulo-4 of the number `$rotnumber` (→ `$modnumber`). Finally, it returns the three values `$rotcode`, `$rotnumber`, `$modnumber`.

For example, in this case, the rotation code `Dp257` would be split into the rotation `Rp` → `$rotcode`, and the number `257` → `$rotnumber`. Since $257 \equiv 3 \bmod 4$, then the value 3 → `$modnumber`.

```
sub rubikmod {

    ## for MODifying (MOD 4)
    print " SUB rubikmod\n";
    ## passing one RotationElement as a parameter, & return a modified one
```

```
## make local variables

my $rot     = "";
my $lencode = "";
my $char    = "";
my $m4      = -1;
my $num     = -1;
my $p       = 0;

# grab the parameter string
# $code  = @_[0];
$code = $_[0];    ## Perl says this is better

$lencode = length $code;

## we want to split the code string into the front (Rubikcode) and terminal number
## so grab 1 char sequentially starting from the end of the string
## and identify the position of the first non-digit char we get to
## example:  $lastchar = substr $code,-1,1 ;
for ( $p = -1 ; $p > -$lencode - 1 ; $p = $p - 1 ) {
    $char = substr $code, $p, 1;
    if ( $char =~ /\d/ ) { }
    else {
        ## this char is the first non-digit from the end"
        ## its position = $p
        last;
    }
}

## now use the value of $p to split the code string
## into front part (= $rot) and back part (= $num)

## get $rot
$rot = substr $code, 0, ( $lencode + $p + 1 );

## get $num
```

```
    $num = substr $code, $lencode + $p + 1, ( $lencode - ( length $rot ) );

    ##--------------
    ## if no number at all (eg D) then this Rubikcode needs to be implemented just once
    ## so allocate its num to have value = 1

    ## if ($num == ""){$num=1}; ## BUT this gives an ErrorMessage when num="" etc
    ## so I have rewritten [if numlength --> 0 then..] then it works OK

    $numlength = ( $lencode - ( length $rot ) );
    if ( $numlength == 0 ) { $num = 1 }
    ##---------------------

    ## determine  mod 4 of the value num
    $m4 = $num % 4;

    ## now return the results
    $rotcode   = $rot;
    $rotnumber = $num;    ## we return this so we can use it as a check
    $modnumber = $m4;

    return $rotcode, $rotnumber, $modnumber;

}    #end of sub

#=====================================
```

3.71 cleanstring

```
sub cleanstring {

    # to clean leading and trailing whitespace from a string
    # from Black Book page 147

    my $line = "";
```

```
    $line = $_[0];    # copied from my RubikMOD()

    #clean leading & trailing whitespace

    $line =~ s/^\s+//;    ## clean leading whitespace
    $line =~ s/\s+$//;    ## clean trailing whitespace

    return $line;
}

#=====================================
```

3.72 cutinfoblock

```
sub cutinfoblock {

    ## remove each  <infoblock> if any exists

    ## pass the whole dataline

    print " SUB cutinfoblock\n";

    my $dataline = $_[0];

    ## we know all brackets are balanced - as this has been checked already.

    print " dataline = $dataline\n";
    my $Langle        = 0;
    my $Rangle        = 0;
    my $angleblock    = "";
    my $lenangleblock = 0;

    ## first see if there is a terminal  infoblock
    $Langle = index $dataline, '<';    ## <
```

```
$Rangle = index $dataline, '>';    ## <
$lenangleblock = $Rangle - $Langle + 1;

##-----------------
## angleblock is the whole block <...> including both angles
## check both angles exist
if ( ( $Langle != -1 ) and ( $Rangle != -1 ) ) {

    my $angleblock = substr( $dataline, $Langle, $lenangleblock );
    print " infoblock(s) present: first =  $angleblock\n";

    my $lenangleblock = length $angleblock;
    my $lendataline   = length $dataline;

    # now need to remove the infoblock from $dataline
    # need to get front and back strings
    my $frontstring    = "";
    my $newfrontstring = "";
    my $backstring     = "";

    $lenbackstring = $lendataline - $Rangle - 1;
    $frontstring = substr( $dataline, 0, $Langle );    # string before Langle
    $backstring = substr( $dataline, $Rangle + 1 );   # string beyond Rangle

    print " Langle possn = $Langle\n";
    print " Rangle possn = $Rangle\n";
    print " lenangleblock (diff + 1) = $lenangleblock\n";
    print " lendataline = $lendataline\n";
    print " lenbackstring = $lenbackstring\n";
    print " frontstring = $frontstring\n";

    #remove the terminal comma from front string
    $newfrontstring = substr( $frontstring, 0, $Langle - 1 );
    print " new frontstring = $newfrontstring\n";
    print " backstring = $backstring\n";
```

```
        ##-------
        # remove angleblock from dataline (join front and back strings)
        $newdataline = $newfrontstring . $backstring;

        $SequenceInfo = substr( $angleblock, 1, $lenangleblock - 2 );

        print " new dataline = $newdataline\n";
        print " SequenceInfo = $SequenceInfo\n";
        print " newdataline = $newdataline\n";
        print " done\n\n";

        return $SequenceInfo, $newdataline;
    }

    else {
        # no infoblock, so need to make newdataline same as orig dataline
        $newdataline = $dataline;
        print " no <infoblock> to remove.\n\n";
        return $newdataline;
    }

    #-----------------

} ## end of sub

#=====================================
```

3.73 fixrepeatelement

```
sub fixrepeatelement {

    print " SUB fixrepeatelement\n";

    print " reformatting any repeat elements...\n";
```

```
## this sub replaces ,-->; and (--> { and ) --> } for the repeat element
## and inserts it back into the original rotation sequence, where it now
## appears as a separate rotation element.

my $repeatstring    = "";
my $lenrepeatstring = "";
my $newrepeatstring = "";
my $frontstring     = "";
my $backstring      = "";
my $p               = 0;
my $q               = 0;
my $len             = 0;
my $k1              = 0;
my $k2              = 0;

## pass the whole dataline without the keyword
my $dataline = $_[0];    # copied from my RubikMOD()

$p = index $dataline, '(';
$q = index $dataline, ')';

print " p = $p, q = $q\n";

$lenrepeatstring = $q - $p + 1;
$repeatstring = substr( $dataline, $p, $lenrepeatstring );

print " first repeat string = $repeatstring\n";
print " length of repeat string = $lenrepeatstring\n";

## translate the chars
$repeatstring =~ tr/,/;/;      ## swap , --> ; Black book page 138--139
$repeatstring =~ tr/\(/\{/;    ## swap ( --> {
$repeatstring =~ tr/\)/\}/;    ## swap ) --> }

$newrepeatstring = $repeatstring;
```

```
    print "...new repeat string = $newrepeatstring\n";

    #------------
    $k1          = $p;                          #start of cut
    $k2          = $p + $lenrepeatstring;       #end of cut
    $frontstring = substr( $dataline, 0, $k1 );
    $backstring  = substr( $dataline, $k2 );
    print " frontstring = $frontstring\n";
    print " backstring = $backstring\n";

    # add insert
    $newdataline = $frontstring . $newrepeatstring . $backstring;
    print " new dataline = $newdataline\n";

    print " done\n\n";

}    # end of sub

#=====================================
```

3.74 repeat

```
sub repeat {

    print " SUB repeat\n";

    ## this SUB expand the repeating elements
    ## this SUB receives a repeat string in the form  {L,R, }3
    ## The original () were converted (above) into {} so we can distinguish the brackets.
    ## we than extract the code sequence and the terminal repeat number
    ## Then we join n copies of the code string to form a long cs string.
    ## then we insert this new long string into the main rotation sequence without the {}
    ## Ultimately the fully expanded rotation sequence is fed into  SUB rotation for processing.

    ## grab the whole repeatstring = {...}n
```

```
my $repeatstring = $_[0];

# the string ={code}n
# get the code sequence and the terminal digit

my $p             = 0;
my $q             = 0;
my $repeatnumber  = 0;
my $repeatcode    = "";
my $lenrepeatcode = 0;

$p = index $repeatstring, '{';
$q = index $repeatstring, '}';
$lenrepeatcode = $q - $p - 1;

$repeatcode = substr( $repeatstring, 1, $lenrepeatcode );
print " repeatcode = $repeatcode\n";    ## correct

$lenrepeatstring = length $repeatstring;

print " lenrepeatstring = $lenrepeatstring\n";
print " lenrepeatcode = $lenrepeatcode\n";
print " p = $p\n";
print " q = $q\n";

##----------------------
## now get the repeat number

if ( $lenrepeatcode == ( $lenrepeatstring - 2 ) ) {
    print " there is no trailing number --> 1\n";
    $repeatnumber = 1;
    print " set repeatnumber = $repeatnumber\n";
}
else {
    $repeatnumber = substr( $repeatstring, $q + 1 );    # correct
    print " repeatnumber = $repeatnumber\n";
```

```
        ## need to check that repeatnumber is a valid integer
        if ( $repeatnumber =~ /\D/ ) {

            # not a valid number

            ## renormalise brackets etc before outputting to LaTeX
            $repeatnumber = restorebrackets($repeatnumber);

            gprint("..*repeat-no. ERROR: $repeatnumber not numeric");
            ErrorMessage(
"repeat-no. $repeatnumber not numeric ?missing comma or nested ()"
            );
        }

    }
    ;    #end of else
    ##-------------------

    ## now make n copies of repeatcode and name the string = $insert
    ## (which is then used by another part of the prog)
    ## we need commas only between elements (not at end)

    $insert = "";            ## $insert = global
    $insert = $repeatcode;
    for ( $t = 1 ; $t < $repeatnumber ; $t = $t + 1 ) {
        $insert = $insert . "," . $repeatcode;
    }

    print " insert = $insert\n";
    print " done\n\n";

}    # end sub

#=====================================
```

3.75 quitprogram

```
sub quitprogram {

    ## exiting the program cleanly
    print " closing down:  writing state...\n";
    writestate();    ## write to the output files
    close;           ## close all files
    exit;
}

#=====================================
```

3.76 showarray

```
sub showarray {

    # show the array as a string

    my @newarray = @_;                          # copied from my RubikMOD()
    my $arraystring = join( ",", @newarray );
    print "  the array  = *$arraystring*\n\n";
}

#=====================================
```

3.77 cleanarray

```
sub cleanarray {

    # cleans array elements of leading and trailing whitespace

    my @cleanset = ();
    my @line     = @_;
```

```
    my $E;

    foreach $E (@line) {
        $E =~ s/^\s+//;    ## clean leading whitespace
        $E =~ s/\s+$//;    ## clean trailing whitespace
        push @cleanset, $E;
    }

    return @cleanset;

}

#=====================================
```

3.78 restorebrackets

```
sub restorebrackets {

    my $line = $_[0];

    ## translate the chars
    $line =~ tr/;/,/;      ## swap , --> ; Black book page 138--139
    $line =~ tr/\{/\(/;    ## swap ( --> {
    $line =~ tr/\}/\)/;    ## swap ) --> }

    return $line;

}

#=====================================
```

3.79 infoblockcolon

```
sub infoblockcolon {
```

```
print "...SUB InfoblockColon\n";

## pass the whole dataline without the keyword
my $line = $_[0];    # copied from my RubikMOD()

if ( ( index $line, '<' ) == -1 ) {

    # no infoblock, so need to make newdataline same as orig dataline
    print " no <infoblock> found.\n\n";
    $newdataline = $dataline;
    return $newdataline;
}
else {
    print " infoblock(s)  present\n";

    print " start-string  = $line\n";

    # look at each char
    my $j    = 0;
    my $char = "";

    my $lenstring = 0;
    $lenstring = length $line;

    # set initial state of inout-flag
    my $inoutflag = "outside";

    #------------------------
    for ( $j = 0 ; $j <= $lenstring ; $j = $j + 1 ) {
        $char = substr( $line, $j, 1 );

        if ( ( $char eq ',' ) and ( $inoutflag eq 'inside' ) ) {

            # replace the char with ;
            substr( $line, $j, 1, ";" );
```

```
        print " colon-string  = $line\n";
    }

    ## need these at end of the loop
    if ( $char eq '<' ) { $inoutflag = "inside" }
    if ( $char eq '>' ) { $inoutflag = "outside" }

};    # end of for

#---------------------------

# -- repeat for [ ] brackets------
$inoutflag = "outside";

for ( $j = 0 ; $j <= $lenstring ; $j = $j + 1 ) {
    $char = substr( $line, $j, 1 );

    if ( ( $char eq ',' ) and ( $inoutflag eq 'inside' ) ) {

        # replace the char with ;
        substr( $line, $j, 1, ";" );
        print " colon-string  = $line\n";
    }

    ## need these at end of the loop
    if ( $char eq '[' ) { $inoutflag = "inside" }
    if ( $char eq ']' ) { $inoutflag = "outside" }

};    # end of for

#---------------------------------
## make an array from the string so we can manipulate the elements
our @linedata = ();
@linedata = split( /,/, $line );

#----------clean the array-------
```

```
        my $E;
        my @cleandata = ();

        foreach $E (@linedata) {
            $E =~ s/^\s+//;    ## clean leading whitespace
            $E =~ s/\s+$//;    ## clean trailing whitespace
            push @cleandata, $E;
        }

        print " colon-array   = @cleandata\n";

        #=================

# Because <infoblocks>  can be located inside curved brackets
# as for example, (\sixspot)2, [\sixspot macro contains an infoblock].
# Consequently, we need to remove
# the <..> blocks as  parts of a string, not as elements in an array.
# --otherwise, removing the terminal infoblock associated with \sixspot
# will result in also removing the right-hand curved bracket --> error.
# So we return the data as a string,  and then send it to sub cutinfoblock later.

        $newdataline = join( ",", @cleandata );
        print "...done\n\n";

        return $newdataline;

    } ## end of else
} ## end of sub

#====================================
```

3.80 RemoveAllSpaces

```
sub RemoveAllSpaces {
```

```
    # remove all spaces in a string
    # from Black book page 143

    my $string = $_[0];

    $string =~ s/\s//g;    # OK

    return $string;

}

#====================================
```

3.81 CheckSyntax

This subroutine is called at an early stage to check the syntax of each input string. After first removing all spaces from the string it then checks for any unbalanced brackets, and also for illegal pairings of characters. If either (a) any bracket is unbalanced, or (b) there are any illegal pairings of characters then the program issues appropriate error messages and then terminates cleanly.

The syntax restrictions are as follows:
(1) all functional elements must be comma separated.
(2) square bracket environment: no commas and no nested brackets
(3) angle bracket environment: no restrictions.
(4) curved bracket environment: no nested brackets, and may end with an optional digit (repeat number).

```
sub CheckSyntax {

    ## this check is used at an early stage in the program, so we can terminate early
    ## if necessary. We check that all () {} <>  are matched (if any exist),
    ##    missing commas, illegal combinations of chars etc.
    ## if any serious errors (eg  brackets not balanced), then we SET an errorflag,
    ##    and terminate the program.

    print " SUB CheckSyntax\n";
```

```
my $dataline = $_[0];

## first clean out all spaces in a string
##   so we can then look for specific combinations of characters
$dataline = RemoveAllSpaces($dataline);

print " dataline = $dataline\n";

##-------------------------------------
## first we check for unbalanced brackets
## count brackets; Angle, Square, Curved
my ( $nleftA,  $nrightA )  = 0;
my ( $nleftS,  $nrightS )  = 0;
my ( $nleftC,  $nrightC )  = 0;
my ( $leftsum, $rightsum ) = 0;

## Blackbook p 139 - counting chars in a string
$nleftA  = ( $dataline =~ tr/</</ );
$nrightA = ( $dataline =~ tr/>/>/ );

$nleftS  = ( $dataline =~ tr/[/[/ );
$nrightS = ( $dataline =~ tr/]/]/ );

$nleftC  = ( $dataline =~ tr/(/(/ );
$nrightC = ( $dataline =~ tr/)/)/ );

print " left and right <> = $nleftA, $nrightA\n";
print " left and right [] = $nleftS, $nrightS\n";
print " left and right () = $nleftC, $nrightC\n";

$leftsum  = $nleftA + $nleftS + $nleftC;
$rightsum = $nrightA + $nrightS + $nrightC;

print " leftsum, rightsum = $leftsum, $rightsum\n";

my $errorflag = "";
```

```
    if ( $leftsum != $rightsum ) {

        if ( $nleftS != $nrightS )
          ## Square brackets
        {
            gprint(
"..*brackets ERROR [ ] Left [$nleftS not equal to Right ]$nrightS"
            );
            ErrorMessage(
                "brackets [ ]: Left [$nleftS not equal to Right ]$nrightS");
            $errorflag = "SET";
        }

        if ( $nleftC != $nrightC )
          ## Curved brackets
        {
            gprint(
"..*brackets ERROR ( ) Left ($nleftC not equal to Right )$nrightC"
            );
            ErrorMessage(
                "brackets ( ): Left ($nleftC not equal to Right )$nrightC");
            $errorflag = "SET";
        }

        if ( $nleftA != $nrightA )
          ## Angle brackets
        {
            gprint(
"..*brackets ERROR < > Left <$nleftA not equal to Right >$nrightA"
            );
            ErrorMessage(
                "brackets < >: Left <$nleftA not equal to Right >$nrightA");
            $errorflag = "SET";
        }
```

```
}

##--------------------------

## check for other bad syntax, eg illegal pairings of characters
## BlackBook p136

my ( $char1, $char2, $charpair ) = "";
my ( $j, $lenstring ) = 0;
$lenstring = length $dataline;
print " lenstring = $lenstring\n";

## we set up a system which allows us to know whether or not we are
## inside a set of brackets. To do this we increment / decrement counters
## each time we pass through a bracket.
## If sum NOT equal to zero, then we are inside etc.

## first initialise each left and right variable.
## seems important that these initialisations are done separately.

my $angleNumLeft  = 0;
my $angleNumRight = 0;
my $angleNumSum   = 0;

my $squareNumLeft  = 0;
my $squareNumRight = 0;
my $squareNumSum   = 0;

my $curvedNumLeft  = 0;
my $curvedNumRight = 0;
my $curvedNumSum   = 0;

## look at each char, and each pair of chars
## with brackets, we increment (right) and decrement (left) the count
## so we can tell if we are inside or outside a set of nested brackets.
## (we need to detect errors in the rotation sequence itself
```

```
    ##  and also inside squarebrackets] since these can occur anywhere,
    ##  but not in the angle infoblocks <..> where we want to be able to write anything)

    for ( $j = 0 ; $j <= $lenstring ; $j = $j + 1 ) {
        $charpair = substr( $dataline, $j,     2 );
        $char1    = substr( $dataline, $j,     1 );
        $char2    = substr( $dataline, $j + 1, 1 );

        ## at top of FOR loop
        if ( $char1 eq '<' ) { $angleNumLeft  = $angleNumLeft + 1 }
        if ( $char1 eq '>' ) { $angleNumRight = $angleNumRight - 1 }

        if ( $char1 eq '[' ) { $squareNumLeft  = $squareNumLeft + 1 }
        if ( $char1 eq ']' ) { $squareNumRight = $squareNumRight - 1 }

        if ( $char1 eq '(' ) { $curvedNumLeft  = $curvedNumLeft + 1 }
        if ( $char1 eq ')' ) { $curvedNumRight = $curvedNumRight - 1 }

        $angleNumSum  = $angleNumLeft + $angleNumRight;
        $squareNumSum = $squareNumLeft + $squareNumRight;
        $curvedNumSum = $curvedNumLeft + $curvedNumRight;

        ##RWDN 22 Oct 2017

        ##------------
        ##  need to trap nested (( )) inside squarebrackets

        if ( $squareNumSum != 0 ) {
            if ( $charpair =~ m/(\(\(|\)\))/ )
            {    # nested curved brackets inside  sq brackets
                gprint("..*syntax error: $charpair -- nested ((..)) in [ ]");
                ErrorMessage(
"$charpair  -- syntax error: nested ((..)) not allowed in [ ]"
                );
                $errorflag = "SET";
            }
```

```
        }
        ##--------------

        ##  if outside angle brackets AND outside square brackets
        ##  then we are checking ONLY the rotation sequence codes

        if ( ( $angleNumSum == 0 ) and ( $squareNumSum == 0 ) ) {

            ##  A-Za-z<    A-Za-z[    A-Za-z(   )A-Za-z    >A-Za-z    ]A-Za-z
            ##   ]<   ][   ](   ])   ]<   )<   )[   )(  ><  >[   >(   d(   d[   d<

            if ( $charpair =~
m/([A-Za-z]\<|[A-Za-z]\[|[A-Za-z]\(|\)[A-Za-z]|\>[A-Za-z]|\][A-Za-z]|\]\<|\]\[|\]\(|\]\)|\]\<|\)\<|\)\[|\)\(|
\>\<|\>\[|\>\(|\d\(|\d\[|\d\<|\[\[|\<\<|\(\(|\)\)|\]\]|\>\>)/
              )
            {
                gprint("..*syntax error: $charpair -- missing comma");
                ErrorMessage("$charpair  -- syntax error:  missing comma");
                $errorflag = "SET";
                next;
            }

            # trap nested curved brackets        (= inside)
            if ( ( $char2 eq "(" ) and ( $curvedNumSum != 0 ) ) {
                ## nested curved brackets
                gprint("..*syntax error: $charpair -- nested ((..))");
                ErrorMessage(
                    "$charpair  -- syntax error: nested ((..)) not allowed");
                $errorflag = "SET";
            }

#-----remove-------------------------------------
# trap comma inside [ ]       ( eq inside)
#        if ($squareNumSum != 0){
#                        if ($char1 eq ",") {
#                            gprint ("..*syntax error: $charpair -- comma not allowed in [ ]");
```

```
#                           ErrorMessage("$charpair  -- syntax error: comma not allowed in [ ]");
#                           $errorflag="SET";
#                           next;
#                         };
#           }; # end of if
#-----------------------------------------------

            ## detect end of string
            if ( $j == $lenstring - 1 ) { last }

        }
        ;    # end of if

    };    # end of for
          #--------------------------

    if ( $errorflag eq "SET" ) {

        ## closing down
        gprint("..*Quiting Perl program -- syntax error");
        ErrorMessage("QUITTING PERL PROGRAM -- syntax error");

        ##------bug fix----------------------
        ## RWDN 5 October 2017
        ## problem = since we are here checking syntax (ie before processing any
        ##   output SequenceXX strings) all four SequenceXX strings will be empty just now.
        ##   This then causes an error if the Rubik user code includes a ShowSequence command,
        ##   since the ForEachX macro used by ShowSequence macro cannot handle an empty string
        ##   when dealing with SequenceShort and SequenceLong.
        ##   So we heve to force these two strings to be just a [\space] before they are output
        ##   by the SUB writestate.
        ##   ie we set  SequenceShort and  SequenceLong strings to \space  here before
        ##   CALLing the SUB quitprogram().
        ##  (to avoid a Rubikcube ShowSequence{}{}{} error if argument is empty
        ##    or is an expandable macro)
        ## This problem arises because the ShowSequence macro uses the ForEachX macro
```

```
        ##   to process each cs element in a string.
        ## Also need to add at least one empty char or comma at end of SequenceLong string,
        ##    as final char (comma) is removed when writing to the out file in SUB writestate
        ##   (CALLed by SUB quitprogram) just prior to closing down.
        ##   NB: if there is no extra terminal char for SequenceLong string,
        ##   then \space --> \spac --> TEX error message
        ##
        ## This issue does not seem to be a problem for SequenceInfo and SequenceName,
        ##    as they  are not returned as cs strings.

        $SequenceShort = "\\space";
        $SequenceLong  = "\\space,";

        ##---------------------------

        print " closing down -- writing state.......... OK\n";

        quitprogram();

    }
    else {
        print " syntax OK; brackets balanced OK\n";
        print " done\n\n";
    }

} ## end sub

#====================
```

3.82 inverse

```
sub inverse {

    my $E = $_[0];
```

```
    my $lastchar   = substr( $E, -1, 1 );
    my $frontchars = substr( $E, 0,  -1 );    # correct

    if ( $lastchar eq "2" ) { $newE = $E }

    elsif ( $lastchar eq "p" ) { $newE = $frontchars }

    else { $newE = $E . "p" }

    return $newE;
}

##======================
##EOF
```

—— END ——