Balíček sudokubundle

Peter Wilson

Abstrakt

Zdrojové kódy v balíku SUDOKU BUNDLE poskytují sadu maker pro zobrazování, řešení a generování her Sudoku. Tento článek popisuje některé pohledy na algoritmy a fungování těchto maker.

Klíčová slova: Sudoku, balíček sudokubundle.

doi: 10.5300/2010-3/227

Introduction

I developed the SUDOKU BUNDLE in response to a challenge presented in the PracTEX journal. It is available from CTAN.ORG with a complete User Manual [8] and is also described in the *LATEX Graphics Companion* [3, Chapter 10].

In December 2005 the *PracTEX Journal* [2] set a competition about Sudoku puzzles. Depending on their experience with TEX, contestants were asked to (a) typeset a particular puzzle, (b) typeset a puzzle described in a 'Sudoku' file, (c) create a solver for Sudoku puzzles. I entered the competition with a printer and solver. Following from this it was no great effort to develop a matching Sudoku puzzle generator. These form the SUDOKU BUNDLE.

A Sudoku puzzle consists of a 9 by 9 grid of cells with some of the cells containing a number between 1 and 9, such as is shown in Figure 1. The problem is to place a number between 1 and 9 in each cell such that no number appears more than once in each row and in each column and in each minor 3 by 3 grid. The solution to the example puzzle is shown later in Figure 4 on page 237. The puzzle and answer have been typeset using the PRINTSUDOKU package.

Among many other sources the *Sudoku Online* [5] website provides much information on Sudoku puzzles and their solutions, as does the *Sudoku Solver by logic* website [6].

A Sudoku puzzle may be represented as a simple text file consisting of nine rows of numbers and dots, nine numbers and dots in each row. The numbers are the clues to the puzzle and the dots represent blanks in the grid. A Sudoku file for the example puzzle is given in Figure 2.

A harder puzzle is in Figure 3. You may like to try and solve it. Whether you do or not, the solution as determined and displayed by the SUDOKU BUNDLE is towards the end of the article in Figure 6 on page 240.

		4	8	3			7	2
	1	2					8	
		5	2		1	3		
				6	2		9	1
7			5		9			3
9	4		7	8				
		3	9		7	4		
	5					6	1	
	8			4	6	9		

Figure 1: Example of simpler Sudoku puzzle

```
                ..483..72
                .12....8.
                ..52.13..
                ....62.91
                7..5.9..3
                94.78....
                ..39.74..
                .5....61.
                .8..469..

              Example of simpler puzzle
(anything can come after the nine puzzle lines)
```

Figure 2: A Sudoku file for the example simpler puzzle (centered)

	3		7			2	9	
2	5	8			1	7		
					5			
		9				8		
			4	2	3			
		2				3		
			8					
		5	6			9	3	7
	9	6			4		8	

Figure 3: A harder puzzle

The SUDOKU BUNDLE only handles sudoku puzzles that consist of 9 by 9 arrays of the numbers 1 through 9. Other puzzles, such as those consisting of 16 by 16 arrays of numbers and letters are outside the scope of this paper.

The SUDOKU BUNDLE consists of three packages:

1. PRINTSUDOKU which prints a puzzle that is contained in an external file or writes out a file that is specifed by a macro in the document.
2. SOLVESUDOKU which solves puzzles up to a certain level of difficulty; it requires the PRINTSUDOKU package to read the puzzle from a file and print it and also to print the solution and write it out to an external file.
3. CREATESUDOKU which generates puzzles that can be solved by SOLVESUDOKU; it requires the SOLVESUDOKU package.

I am not going to describe the packages in detail as you can find that elsewhere [8]. Rather, I shall discuss some of the algorithms that I used in the packages and how I implemented them in LaTeX.

Typesetting

This is the province of the PRINTSUDOKU package, which provides two basic functions:

1. The `\sudoku{`⟨*file*⟩`}` macro reads a Sudoku game from ⟨*file*⟩ and typesets the grid and clues;
2. The `\writepuzzle{`⟨*line1*⟩`}{`⟨*line2*⟩`}`...`{`⟨*line9*⟩`}[`⟨*text*⟩`]` macro writes the nine lines of a puzzle to the `\puzzlefile` external file, where the default file is: `\newcommand*{\puzzlefile}{puzz.sud}`

The code for `\writepuzzle` is pretty simple, just opening an output file and writing the 9 arguments to the file a line at a time:

```
\newwrite\s@dwrite
\newcommand*{\writepuzzle}[9]{%
  \immediate\closeout\s@dwrite
  \immediate\openout\s@dwrite=\puzzlefile
  \immediate\write\s@dwrite{#1}%
  ...
  \immediate\write\s@dwrite{#9}%
  \writes@dpuzzend}
```

`\writes@dpuzzend` provides an option to write text after the puzzle data is written, with `\sudpuzznewline` as a new line macro (`\\` will not work here).

```
\newcommand*{\writes@dpuzzend}[1][\@empty]{%
  \ifx\@empty #1\else
    \immediate\write\s@dwrite{ }% a blank line
    \immediate\write\s@dwrite{#1}%
  \fi
  \immediate\closeout\s@dwrite}
\newcommand*{\sudpuzznewline}{^^J}
```

The `\writepuzzle` and `\writes@dpuzzend` pair of macros is a particular instance of a general technique for making it appear that a macro takes more than the 9 argument limit imposed by TEX.

More complicated is the code for the `\sudoku{`⟨*file*⟩`}` macro, which reads a puzzle from the ⟨*file*⟩ and typesets it.

```
\newcommand*{\sudoku}[1]{%
  \reads@dgame{#1}% open the file for reading
  \s@dgame}
```

The macro `\s@dgame` uses the `picture` environment to draw the grid. It then inserts the clues into the grid.

```
\newcommand*{\s@dgame}{%
  \setlength\unitlength\halfs@dcell % units of half cell size
  \begin{picture}(18,18)(0,-18)
%% code to draw the grid
  \adds@dclues
  \end{picture}}
```

The macro `\adds@dclues` (not shown) reads the puzzle file line by line, each time setting `\firsts@dcluetrue`, and then for each line calls `\dos@dcols` to insert its clues into the grid.

```
\newcommand*{\dos@dcols}{%
\bgroup
  \loop%        over the 9 clues
    \ifnum\s@dncol<10\relax
%% calculate grid location coordinates (\s@dcolpos,-\s@drowpos)
    % put the clue into the grid
    \put(\s@dcolpos,-\s@drowpos){\makebox(0,0){\gets@dclue}}%
    \advance\s@dncol 1\relax% increment clue/column count
  \repeat
\egroup}
```

The macro `\gets@dclue` retrieves the next clue (character) from the line of clues and presents it for printing. To do this it uses `\splitoff{`⟨*string*⟩`}`, which gets the next character in a string, making it available as `\istchar` and leaves the remainder of the string as `\restchars`. The technique is based on TEX's delimited arguments [4, chapter 10]. I have talked about this in more detail in two of my *Glisterings* columns [7, 9], the second of which also contains a long example of using the `\loop...\repeat` construct.

```
\def\gettwo#1#2\nowt{%
  \gdef\istchar{#1}\gdef\restchars{#2}}
\def\splitoff#1{\gettwo#1\nowt}
```

Finally, this is the code for `\gets@dclue`. If the clue is a number, `\gets@dclue` provides it for printing or if it is a `'.'` then `\gets@dclue` skips on. At the beginning the string is the line as read from the file (`\s@dline`); after that the string is `\restchars`.

```
\gdef\s@dfstop{.}
\newcommand*{\gets@dclue}{%
  \iffirsts@dclue%  initially set by \adds@dclues
    \expandafter\splitoff\expandafter{\s@dline}%
    \global\firsts@dcluefalse
  \else
    \expandafter\splitoff\expandafter{\restchars}%
  \fi
  \ifx\s@dfstop\istchar%  a '.' return nothing
  \else%                  return clue number
    \istchar
  \fi}
```

Solving

The `\sudokusolve{`⟨*file*⟩`}` macro in the SOLVESUDOKU package attempts to solve the puzzle contained in the ⟨*file*⟩. It first prints the puzzle as specified in ⟨*file*⟩, then solves it as best it can, and lastly typesets the (partial) solution.

The following facts are used to generate a solution.

1. Initially the potential solution for any cell is in the set of digits 1...9.
2. In a solved puzzle a digit must be unique within its row, its column, and its 3 by 3 block. So, if a solution, say N, is known for a cell, then N can be deleted from all the potential solutions in the other cells of the row, column and block. I have called this a *simple reduction.*
3. If among all the cells in a row (column, block) there is a digit that occurs only once among all the potential solutions, then that digit is the solution for its cell. I have termed this a *loner.*
4. If among all the cells in a row (column, block) there are two digits which occur only twice in the potential solutions, each time as a pair (e.g., 39 and 39), then one or other of the two digits must be a solution for a cell in which the pairs occur. This means that the two digits cannot occur anywhere else in the row (column, block) and thus can be eliminated from all the other potential solutions. I call this *pair reduction.*
5. There are other facts which are more difficult to apply and I have not considered them because of their complications and the difficulty of embodying them in LaTeX code.

The solution procedure is:

1. Populate the puzzle grid by assigning to each cell either the clue (digit) given in the puzzle ⟨*file*⟩ or the set of potential solutions when the puzzle provides no value.
2. For each clue perform a simple reduction, which may produce loners.
3. Continue the simple reductions until there is no change in the solution state. This is either because the full solution has been obtained or that more sophisticated methods are needed.
4. Examine the partial solution for pairs and if one is found perform the pair reduction. After a pair reduction go back and look for loners (and subsequent simple reductions (and subsequent pair reductions). At the end either a complete solution is found or the solver gives up.
5. The process stops when either all 81 cells have been solved or there is no change in any potential solution after going through all the reductions.

The major problem was in deciding on a convenient datastructure for the problem. In the end I used a 9 digit 'binary solution set' for the representation of a cell's potential solution (e.g., [111111111] $\Leftrightarrow$ 123456789 and [101010101] $\Leftrightarrow$ 13579). The solution, say N, for a cell is represented as the 'set' [$-N$]; that is, for example, a potential solution '3' is represented as [001000000] and the actual solution '3' is represented as [-3]. I will use the term *9-set* to indicate a set with a maximum of 9 members, where a member is a digit d in the range $1 \leq d \leq 9$.

Following from this was the question of how to implement the datastructure? There are 81 cells in the Sudoku grid and I needed to maintain a potential or actual solution for each cell. It was convenient to use a `\count` for each cell's solution set which was accessible via the cell's number (1. . . 81).

```
\newcommand*{\newknt}[1]{\expandafter\newcount\csname #1\endcsname}
\newcommand*{\useknt}[1]{\csname #1\endcsname}
```

`\newknt{`⟨*id*⟩`}` creates a new `\count` called ⟨*id*⟩, where ⟨*id*⟩ can include analphabetic characters (like digits), and `\useknt{`⟨*id*⟩`}` expands to the ⟨*id*⟩ `\count` created previously by `\newknt`.

```
% make the potential solution sets
\newcommand*{\makesudsets}{%
  \global\s@lcnta=1\relax
  \loop
    \ifnum\s@lcnta<82\relax
    \newknt{s@lans\the\s@lcnta}%
    \global\useknt{s@lans\the\s@lcnta}=111111111\relax
    \advance\s@lcnta 1\relax
  \repeat}
```

`\makesudsets` creates 81 `\count`s named `\s@lans1` through `\s@lans81` and sets them all to 111111111.

Now some macros are needed to manipulate a 9-set. These are principally based on the fact that TeX only provides integer arithmetic. For instance, with integer arithmetic

$$19/10 = 1,\ 20/10 = 2 \text{ and } 21/10 = 2,$$

which is a method for getting the first digit of a two-digit number (and similarly for numbers with more digits). Further,

$$(19/10) \times 10 = 10 \text{ while } 19 - (19/10) \times 10 = 9$$

which provides a method for obtaining the last digit of a two-digit number. As a more complicated example, to determine the number of thousands in a number, say 13247546 where the answer is 7,

$$\begin{aligned}(13247546/1000) &= 13247\\ (13247/10) \times 10 &= 13240\\ 13247 - 13240 &= 7\end{aligned}$$

`\settonum{`⟨*set*⟩`}{`⟨*cnt*⟩`}` converts a potential binary solution 9-set ⟨*set*⟩ to the corresponding list of digits, e.g., `[11...1] -> 12...9`. The result is assigned to the `\count` ⟨*cnt*⟩ which must be supplied by the calling macro. If the set is negative then the result is that number (e.g., `[-3] -> -3`). If the set contains only a single non-zero entry, that is converted to the negative of the corresponding digit (e.g., `[100] -> -7`).

```
\newcommand*{\settonum}[2]{%
  \settonumcnt=#1\relax
  \tempcnty=0\relax
  \tenscnt=1\relax
  \ifnum\settonumcnt<0\relax % just return the number
    \tempcnty=\settonumcnt
    #2=\tempcnty
  \else
    \ifodd\settonumcnt  % set is [dddddddd1] so 9 flagged
      \tempcntz=9\relax
      \multiply\tempcntz \tenscnt
      \advance\tempcnty by \tempcntz
      \multiply\tenscnt 10\relax
    \fi
    \divide\settonumcnt by 10\relax % set reduced to [dddddddd]
    \ifodd\settonumcnt  % reduced set is [ddddddd1] so 8 flagged
      \tempcntz=8\relax
      \multiply\tempcntz \tenscnt
      \advance\tempcnty by \tempcntz
      \multiply\tenscnt 10\relax
    \fi
    \divide\settonumcnt by 10\relax % set reduced to [ddddddd]
    \ifodd\settonumcnt  % reduced set is [dddddd1] so 7 flagged
      ...
    \ifodd\settonumcnt  % reduced set is [1] so 1 flagged
      \tempcntz=1\relax
      \multiply\tempcntz \tenscnt
      \advance\tempcnty by \tempcntz
    \fi
    \ifnum\tempcnty<10\relax
      \ifnum\tempcnty>0\relax % single digit
        \tempcnty = -\tempcnty
      \fi
    \fi
    #2=\tempcnty
  \fi}
```

\numofnuminset{⟨*dig*⟩}{⟨*set*⟩}{⟨*cnt*⟩} sets the \count ⟨*cnt*⟩ to the number of times the digit ⟨*dig*⟩ is represented in the 9-set ⟨*set*⟩. For example the number of the digits in the 9-set [200000013] are 1->2, 2->0, ..., 8->1 and 9->3.

```
\newcommand*{\numofnuminset}[3]{%
  \tempsetctr=#2\relax
  \tempsetansrctr=\tempsetctr
  \ifnum\tempsetctr<0\relax % a solution, not a set
    \tempsetansctr=0\relax
  \else
    \ifcase #1\relax
    \or     % 1
      \divide\tempsetansctr by 100000000\relax
    \or     % 2
      \divide\tempsetansctr by 10000000\relax
      \tmpsetctr=\tempsetansctr
      \divide\tmpsetctr 10\relax \multiply\tempsetctr 10\relax
      \advance\tmpsetansctr -\tmpsetctr
    \or    % 3
      \divide\tempsetansctr by 1000000\relax
      \tmpsetctr=\tempsetansctr
      \divide\tmpsetctr 10\relax \multiply\tempsetctr 10\relax
      \advance\tmpsetansctr -\tmpsetctr
    \or
      ...
    \or   % 9
      \tmpsetctr=\tempsetansctr
      \divide\tmpsetctr 10\relax \multiply\tempsetctr 10\relax
      \advance\tmpsetansctr -\tmpsetctr
    \else % error
      \tmpsetansctr=0\relax
    \fi
  \fi
  #3=\tmpsetansctr}
```

The macro `\deletenumfromset{`⟨*dig*⟩`}{`⟨*set*⟩`}{`⟨*cnt*⟩`}` removes the digit ⟨*dig*⟩ from the potential binary solution 9-set, putting the modified set in `\count` ⟨*cnt*⟩. If the digit was removed then the boolean `\ifsetchanged` is set TRUE.

```
\newcommand*{\deletenumfromset}[3]{%
  \global\setchangedfalse
  \tmpsetctr=#2\relax
  \tmpsetansctr=#2\relax
  \ifnum\tmpsetctr<0\relax% represents a solved number, do nothing
  \else
    \ifcase #1\relax
    \or%             1
      \divide\tmpsetctr by 100000000\relax
      \ifodd\tmpsetctr%  it's there
        \advance\temsetansrctr -100000000\relax
        \global\setchangedtrue
      \fi
    \or%            2
      \divide\tmpsetctr by 10000000\relax
      \ifodd\tmpsetctr%  it's there
        \advance\temsetansrctr -10000000\relax
        \global\setchangedtrue
      \fi
    \or
      ...
    \or%            8
      \divide\tmpsetctr by 10\relax
      \ifodd\tmpsetctr%  it's there
        \advance\temsetansrctr -10\relax
        \global\setchangedtrue
      \fi
    \or%            9
      \ifodd\tmpsetctr%  it's there
        \advance\temsetansrctr -1\relax
        \global\setchangedtrue
      \fi
    \fi
  \fi
  #3=\tmpsetansctr}
```

Given these macros it is now just a case of using them in a lot of tedious code to solve the puzzle using the procedure described earlier.

Going through the puzzle as presented, for any cell for which a clue is given, its potential binary solution set is replaced by the actual solution. After this initialisation, `\deletenumfromset` is used for the simple reductions.

The macro `\numofnuminset` is used to find loners — the potential binary solution 9-sets for the cells in a row (column, block) are added together and the result is then searched for a loner, which is a digit that occurs only once in the resulting 9-set summation.

The macro `\numofnuminset` is also used to search for pairs in a row (column, block) following a similar summation of the potential binary solution 9-sets.

`\settonum` is used to determine if a cell's 9-set has been reduced to a single digit, which is then a solution. It can alo be used in printing out the current status at any point during the solution process, showing for each cell either the solution or the list of potential solutions for that cell.

For further details look at the documented code — there's only about 1400 lines of it — for the SOLVESUDOKU package.

Towards the end of this article the solution to the puzzle presented in Figure 3 found by `\sudokusolve` is provided as Figure 6 on page 240. The figure is produced by the following code:

```
\begin{figure}
\centering
\cluefont{\normalsize}\cellsize{1.5\baselineskip}
\sudokusolve{cal4s4.sud}
\caption{The puzzle from \fref{fig:puz2} together with its solution
         as found and presented by \cs{sudokusolve}}
         \label{fig:ans2}
\end{figure}
```

where the `\cs` macro is defined as

```
\DeclareRobustCommand\cs[1]{\texttt{\char`\\#1}}
```

which is most useful if you ever need to typeset the name of a macro.

Generating

The CREATESUDOKU package lets you automatically create Sudoku puzzles of the kind that the SOLVESUDOKU package, which it uses, can solve. It also uses Donald Arseneau's RANDOM.TEX for generating random numbers [1].

The package requires a completely solved puzzle to start with, which can be either from a file that you provide, or it uses a default solved puzzle.

The starting grid is modified in a random manner. Within one of the three columns of blocks exchanging any two of the three cell columns alters the puzzle

6	9	4	8	3	5	1	7	2
3	1	2	6	7	4	5	8	9
8	7	5	2	9	1	3	6	4
5	3	8	4	6	2	7	9	1
7	2	6	5	1	9	8	4	3
9	4	1	7	8	3	2	5	6
1	6	3	9	5	7	4	2	8
4	5	9	3	2	8	6	1	7
2	8	7	1	4	6	9	3	5

Figure 4: Solution to the simpler example puzzle in Figure 1

but leaves it still as a valid result. For example columns 1 and 3 (in the first column of blocks) may be exchanged and columns 8 and 9 (in the third column of blocks) be exchanged and the result is still a solved grid. Similarly, within a row of blocks, exchanging any two of the three rows of cells changes the puzzle but leaves it as a valid result.

The default starting grid is shown in Figure 5. Check that it is a valid Sudoku solution and then try exchanging pairs of rows and columns, as described above, to check that the result, although different, is still a valid solution.

Row pairs and column pairs are exchanged in a random fashion a random number of times. At this point the grid is a complete solution. There is a macro that will randomly eliminate 17 numbers from the grid; a puzzle is ambiguous, that is it has more than one solution, if two numbers are completely absent from the grid. You can then get the package to delete particular numbers, rows, columns, blocks, or diagonals from this grid.

When given its head the package writes the puzzle to an external file `\prevfile` and then uses `\sudokusolve` to try and solve the puzzle. If it can not find a solution then you would have to go back and try again, eliminating fewer and/or different clues. If `\sudokusolve` can solve the puzzle the package randomly eliminates a clue, writes the revised puzzle to another external file (`\currfile`) and uses `\sudokusolve` to try and solve the new puzzle. If it can then the

1	2	3	4	5	6	7	8	9
4	5	6	7	8	9	1	2	3
7	8	9	1	2	3	4	5	6
2	3	4	5	6	7	8	9	1
5	6	7	8	9	1	2	3	4
8	9	1	2	3	4	5	6	7
3	4	5	6	7	8	9	1	2
6	7	8	9	1	2	3	4	5
9	1	2	3	4	5	6	7	8

Figure 5: Default initial puzzle for generation

`\currfile` puzzle is written into the `\prevfile`, another clue is randomly eliminated, and the process continues on. At the point where `\sudokusolve` fails, the last succesfully solved puzzle is that in the `\prevfile`. This is the puzzle that is presented as the newly created puzzle.

An interactive program

For checking how well the `\sudokusolver` was working I wrote a small interactive LaTeX program that asked for a Sudoku puzzle file to solve, tried to solve it, and kept on asking for more files until I effectively said 'no more'.

Here is a version of it, which I have called 'solvem'. To use it simply call:
`pdflatex solvem`
and a session will look like this, where, for exposition purposes, the user's commands/responses are typeset in *this font* and `solvem`'s are in `this font`, and the user wants solutions to the puzzles in the puzzle files `cal4s4.sud` and `st123.sud`:

```
    pdflatex solvem
    New file?  y/n
    \getans=y
    Enter the file name
```

```
\sudfile=cal4s4.sud
...progress report on the solution...
New file?  y/n
\getans=y
Enter the file name
\sudfile=st123.sud
...progress report on the solution...
New file?  y/n
\getans=n
Output written on solvem.pdf
```

Here is the program:

```
% solvem.tex   Solve Sudoku
%              author Peter Wilson
\documentclass{article}
\usepackage{solvesudoku}
\newcommand*{\solvefile}[1]{%
\begingroup
  \sudokusolve{#1}%
  \par
  \vspace{\baselineskip}%
  Number of clues = \the\numcluesctr\ and difficulty = \the\difficultyctr.
\endgroup}
\def\yesans{y}
\begin{document}
\loop
  \typein[\getans]{New file? y/n}
  \ifx\yesans\getans
    \typein[\sudfile]{Enter the file name}r
    \IfFileExists{\sudfile}{%
    \clearpage
    \begin{center}\huge \sudfile\end{center}
    \solvefile{\sudfile}%
    }{\typeout{I can't find file \sudfile}}
\repeat
\end{document}
```

The command `\typein[`⟨*csname*⟩`]{`⟨*text*⟩`}` is a LaTeX macro that outputs ⟨*text*⟩ to the terminal and `.log` file and waits for some response text which it assigns to the command ⟨*csname*⟩ and you can then do something with `\csname`. It is an extended version of `\typeout{`⟨*text*⟩`}` which just outputs ⟨*text*⟩ to the terminal and the `.log` file.

Basically `solvem.tex` goes round a loop asking for a puzzle file and calls `\sudokusolve` to try and solve it. There is code to catch if a file specified by the

	3		7			2	9	
2	5	8			1	7		
					5			
		9				8		
			4	2	3			
		2				3		
			8					
		5	6			9	3	7
	9	6			4		8	

THE ANSWER

6	3	1	7	4	8	2	9	5
2	5	8	9	6	1	7	4	3
9	7	4	2	3	5	6	1	8
3	4	9	5	7	6	8	2	1
8	1	7	4	2	3	5	6	9
5	6	2	1	8	9	3	7	4
1	2	3	8	9	7	4	5	6
4	8	5	6	1	2	9	3	7
7	9	6	3	5	4	1	8	2

Figure 6: The puzzle from Figure 3 on page 228 together with its solution as found and presented by `\sudokusolve` (design of the output is slightly modified for the sake of the article)

user does not exist, in which case it asks for another one. After each solution it gives some information about how hard it was to solve (or not as the case may be) the puzzle. The puzzles and their solutions are written to the output file from `solvem.tex`, either `solvem.dvi` or `solvem.pdf`, depending on how `solvem` is called.

References

[1] Donald Arseneau. Generating random numbers in TEX, 1995. Available on CTAN.ORG in `macros/generic/misc/random.tex`.

[2] The Editors. Distractions: Sudoku. In *The PracTEX Journal*, Volume 1, Number 4, 2005. ISSN 1556-6994. Available at `http://tug.org/pracjourn/2005-4/distract/`.

[3] Michel Goossens, Frank Mittelbach, et al. *The LATEX Graphics Companion*. Addison-Wesley, 2nd edition, 2008. ISBN 0-321-50892-0.

[4] Donald Knuth. *The TEXbook*. Addison-Wesley, 1986. ISBN 0-201-13448-0.

[5] Sudoku Online: Home of the Sudokulist. `http://www.sudoku.org.uk/`

[6] Sudoku Solver . . . by logic. `http://www.sudokusolver.co.uk/`

[7] Peter Wilson. Glisterings. In *TUGboat*, Volume 26, Number 3, pp. 253–255, 2005. ISSN 0896-3207. Available at `http://tug.org/TUGboat/Articles/tb26-3/tb84glister.pdf`.

[8] Peter Wilson. The sudoku bundle for displaying, solving and generating Sudoku puzzles, 2006. Available on CTAN.ORG in `macros/latex/contrib/sudokubundle/`.

[9] Peter Wilson. Glisterings: stringing along, loops. In *TUGboat*, Volume 28, Number 1, pp. 12–14, 2007. ISSN 0896-3207. Available at `http://tug.org/TUGboat/Articles/tb28-1/tb88glister.pdf`.

Summary: The sudoku bundle

The SUDOKU BUNDLE provides a coordinated set of packages for displaying, solving, and generating Sudoku puzzles. This article describes some of the internal aspects of the packages.

Keywords: Sudoku, sudokubundle package.

*Peter Wilson, **herries.press@earthlink.net***
CSTUG c/o FEL ČVUT, Technická 2
Prague, CZ-166 27, Czech Republic