Shuffle: a program to randomize lists with optional sequential constraints

Christophe Pallier
INSERM U562, SHFJ, Orsay, France

Address for correspondence: Christophe Pallier, INSERM U562, SHFJ, 4 pl. du Gal Leclerc, Orsay, F91401 FRANCE. Email: pallier@lscp.ehess.fr; tel: 33 1 69 86 78 73; fax: 33 1 69 86 78 16

`shuffle` is a program used to extract random samples or to produce "quasi-randomized" sequences of items. It allows to select permutations that avoid repetitions of similar items: The user can set limits on the number of consecutive occurrences of items associated with some labels, or, can specify the minimal distance separating two items with the same labels. `shuffle` is useful for extracting random samples from a dictionary, or for generating quasi-randomized lists of stimuli that avoid runs of trials belonging to the same categories.

Introduction

Experimental psychologists often need to extract random samples from lists of items. For example, psycholinguists typically select subsets of words from dictionaries. The sampling must be done randomly to allow statistical inference. Another situation which requires randomization happens when the experimenter needs to shuffle the order of trials differently for each participant in an experiment. Shuffling and random sampling are related because extracting a random sample of n items from a list is equivalent to shuffling the list and selecting the first n items from the result.

Shuffling the order of trials for each participant aims to diminish the impact of position-specific effects and of between-trials interactions. Such interactions can arise when series of trials contain long sequences with similar characteristics; for example, successive trials may contain stimuli that have similar features, or may map systematically onto the same response. It is known that response times of participants are sensitive to such local repetitions (Luce, 1986). Therefore, an experimenter may want to avoid long runs of similar trials. In some cases, such as in long-term repetition priming experiments, it is even necessary that similar items be separated by a minimum number of intervening trials.

This paper presents `shuffle`, a program designed to extract random samples and to 'quasi-randomize' the order of items in a list while avoiding long runs of similar items, or while imposing a minimum interval between instances coming from specific categories of items.

Using 'shuffle'

The package "shuffle" contains two programs: `shuffle.pl` and `shuffle_tk.pl`. The first is to be used on the command line, while the second provides a graphical interface. New users will probably prefer the version with a graphical interface. The command line version, however, has the advantage that it is faster to run, and can be used as a building block in more complex programs; it will therefore appeal more to advanced users. Figure 1 shows the interface of shuffle_tk.pl.

The main part of the window consists of a text area where the list of items can be edited directly or can be pasted from the clipboard. The list can also be read from a text file with the 'Open File' button. The items in the list must appear on successive lines. Pressing the 'Shuffle!' button shuffles the lines from the text area, applying a random permutation to the original lines.

To output only a random subset of the original list, the number of desired items must be entered in the "N=" box. This shuffles the whole list, yet displays only the first "N" items from the results.

Another option, the "seed", can be used to enter a number that is used to initialize the random number generator; this number thus characterizes the permutation generated. The same number can be used to apply the same permutation to several lists.

When the "Constraints" box is empty, all permutations are allowed and are equiprobable, meaning that any of the potential $n!$ permutations has a $1/n!$ probability of being applied. To set constraints on permissible permutations, it is possible to enter a series of numbers in the "constraint" box. This option is meaningful only when the input text is in tabular format, that is when each line contains the same number of tokens separated by spaces (as in figure 1). The tokens in some or all of the columns may be considered as "labels" corresponding to levels of experimental factors. It is then possible to reject permutations in which the labels are repeated across successive lines more than a fixed number of times.

The i^{th} number in the constraint box corresponds to the i^{th} column in the input. When set to '0', there is no constraint

E-mail: `pallier@lscp.ehess.fr`. Web: `http://www.pallier.org`

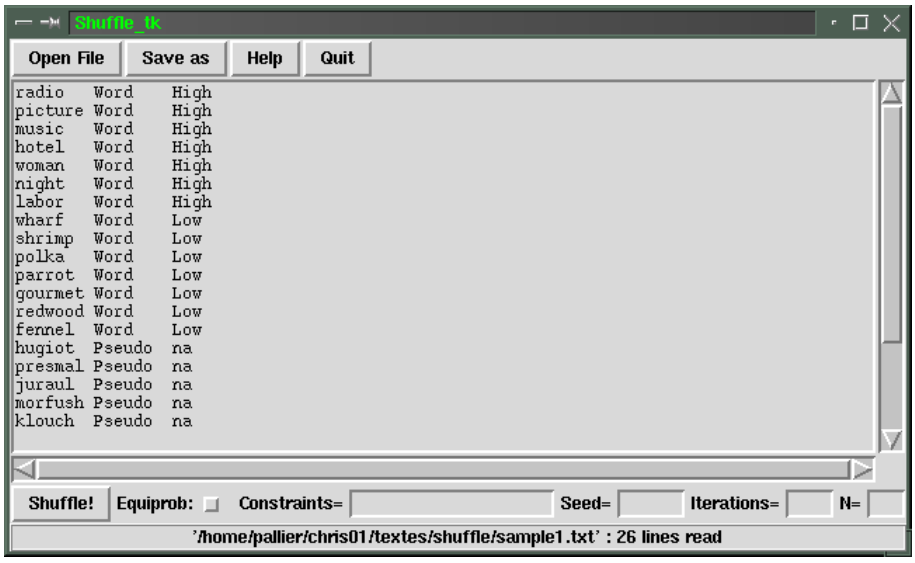

Figure 1. Main window of shuffle_tk

in the relevant column. When set to a positive number, this number indicates the maximum number of repetitions in the corresponding column (across successive lines). When set to a negative number, e.g. $-n$, this indicates that a least n lines must separate two occurrences of the same label. For example, if the second column of the list contains a label 'Word' or 'Pseudo', as in the list shown in Figure 1, one can avoid permutations in which more than four consecutive lines contain the same label by setting the constraints to '0 4'.

Using negative numbers as constraints is useful, for example, to generate lists where a small number of targets are spread among a larger number of filler items: a column could be created containing either the label 'TARGET' or 'FILLERxxx' where 'xxx' varies for each filler. If the constraint '-10' is associated with this column, targets will be separated by at least 10 fillers.

There are actually two different algorithms to generate constrained permutations. The first is a very simple one: it randomizes the lists and checks whether the result respects the constraints. If not, the permutation is discarded and a new one is tried (The maximum number of trials can be specified with in the 'iter' box).

The second algorithm is more clever: it also starts by randomizing the order, but then, it scans the result line by the line and rearranges the order when any of the constraints is violated. This algorithm almost always find a solution, even when the constraints put very strong restrictions on permissible permutations.

Both algorithms are provided because the first one guarantees an equiprobable draw from the set of all permissible permutations, which is sometimes desirable. However, it will often fail to find a suitable solution. This is why the second algorithm is used by default. To select the simple algorithm, the option 'Equiprob' must be checked.

The command line version of the program, "shuffle.pl", has the same options as the graphical version, but they are passed as arguments on the command line. Listing 1 shows a few examples of its usage. Note that in these examples, 'shuffle' takes its input from the file 'sample.txt' and prints the result on the terminal. To save it in a file, the redirection operator '>' must be used.

Proof of equiprobability

The pseudo-code for the generation of an unconstrained random permutation fits in two lines:

```
Read line[1] to line[N]
For i := N downto 2 do swap( line[i], line[random(i)] )
```

The function `random` returns a random number between 1 and its argument. The function `swap` exchanges two lines.

We are going to use the recurrence principle to demonstrate that this algorithm makes all the permutations of the lines equiprobable (that is, each line has a probability of $1/n$ of appearing with any of the output rank). It helps to note that the previous code is the iterative equivalent of the following recursive algorithm:

```
function perm(n) {
 swap(line[n],line[random(n)])
 if n>1 then perm(n-1)
}
```

For $n = 2$, line[2] is swapped either with line[1] or with line[2] (itself), each case having probability 1/2. Therefore the two permutations, (1,2) and (2,1), are equiprobable.

Let us suppose that the algorithm generates equiprobable permutations up until rank $n-1$ (that is, any given line has probability $1/(n-1)$ of occurring in any given output rank, for a permutation of size $n-1$). To generate a permutation of rank n, the algorithm first swaps the n^{th} line randomly

Figure 2. Listing 1: a few examples of shuffle's use

```
shuffle -?               # to display a short help
shuffle sample.txt
shuffle -n10 sample.txt  # limits output to 10 lines
shuffle -n10 sample.txt  # new permutation of 10 lines
shuffle -n5 -s134 sample.txt # sets the seed of the random generator
shuffle -n5 -s134 sample.txt # you get the same permutation as before...
shuffle -n8 -c'2' sample.txt # no more than 2 successive lines with
                             # the same label in column 1
shuffle -n8 -c'0 3' sample.txt # no more than 3 successive lines with
                               # the same label in column 2
shuffle -n8 -c'2 2' sample.txt # no more than 2 successive lines with
                               # the same label in column 1 or column 2
shuffle -n8 -c'1 1' sample.txt # not much room for randomness
shuffle -e -c'1 2' sample.txt  # use the 'equiprob' algorithm
```

with one of the lines between 1 and n, therefore each line has the same probability, $1/n$, of occurring in output rank n. Then the algorithm applies itself recursively to the $n-1$ first elements; the recurrence hypothesis applies: all $n-1$ lines have an equal probability $1/(n-1)$ of being assigned any given output rank between 1 and $n-1$. Therefore, a given line (with rank between 1 and n) has a probability of $(n-1)/n \times 1/(n-1) = 1/n$ to be assigned to any given output rank between 1 and $n-1$.

Algorithm for constrained permutations

As said earlier, `shuffle` provides two algorithms to generate constrained permutations. The first simply filters the output of the unconstrained algorithm: permutations are generated until one is found that fulfills the constraints or until a threshold on the maximum number of trials is reached. This random search in the space of permutations is not efficient, but it guarantees the absence of bias in the selection of the constrained permutation.

The second algorithm generates a constrained permutation similarly to the way in which a human would do it by hand. First, an unconstrained permutation of the input lines is generated; the algorithm then checks, from the first line downward, whether of not all constraints are fulfilled. Every time a line is found that violates a constraint, the algorithm searches systematically for an acceptable line further down the list. If such a line is found, the algorithm swaps it with the line violating the constraint. If no acceptable line can be found, the current permutation is abandoned and a new one is tried. Our experience is that this algorithm find suitable solutions even in when the constraints are very strong. For example, if a column contains 50 labels 'Words' and 50 labels 'Pseudo' and the corresponding constraint is set to 1, then the only acceptable permutations must alternate the labels. While the first algorithm has practically no chance of finding any such permutation, the second will find one without difficulty.

Installing shuffle

The "shuffle" package, in zip format, can be downloaded from the software page on the author's web site (http://www.pallier.org/softs.htm). It contains two programs: `shuffle.pl` and `shuffle_tk.pl`. The first is meant to be used on the command line and the second provides a graphical interface. The package is distributed under the terms of the GNU General Public License 2 (GPL, 1991), allowing anybody to modify the source code. The programs are written in Perl (Schwartz & Phoenix, 2001). Therefore, before using them, an interpreter for the language Perl must be installed on your computer; moreover the graphical version requires the Tk module for Perl (Walsh, 1999). Below, we provide the installation directions for Linux/unix and for Windows.

Linux/unix

Perl is pre-installed in most Linux/unix distributions; otherwise, it can always be downloaded the "Comprehensive Perl Archive Network" (http://www.cpan.org). To use the graphical version, you may have to install the Tk module. To check whether it is installed on a system, try to run 'Perl -MTk': in case an error message is issued, the module is not installed. It can be installed by the superuser by calling 'perl -MCPAN -e shell' and typing 'install Tk;' (see 'man CPAN' for detailed information on installing perl modules).

Copy `shuffle.pl` and `shuffle_tk.pl` in a directory listed in the PATH variable (e.g. `~/bin`). You may need to edit the first line of both programs to reflect to actual location of perl on the system (as revealed by the command 'which perl'). The default version of the files assumes that the perl program is located at `/usr/bin/perl`, but you may, for example, have to change this to `/usr/local/bin/perl`. Finally, you can create a link to shuffle.pl in the same directory with the command "ln -s shuffle.pl shuffle".

Mac OS-X is based on a unix variant, BSD, and perl comes installed by default. The command line version therefore works quite well. However, the installation of the Tk module, which is necessary for the graphical version, though possible, is currently (in the spring

2002), quite involved and reserved to advanced users (cf. http://www.lehigh.edu/~sol0/Macintosh/X/ptk.)

Windows

A free Perl interpreter which includes the Tk module is available at http://www.activestate.com/Products/ActivePerl. The download is about 8Mb. Windows 95 users also have to download the Microsoft Installer version 2. Once Perl is setup, to install the "shuffle" package, you need to:

1. Copy `shuffle.pl` and `shuffle_tk.pl` in a directory listed in the PATH variable, e.g. `c:/Perl/bin`.

2. Create a new shortcut on the desktop, associated with the command 'perl c:/perl/bin/shuffle_tk.pl'. Clicking on the associated icon will then launch the graphical program immediately.

3. In using the command line version, to avoid typing"perl c:/perl/bin/shuffle.pl" all the time, you can define an alias with the program 'doskey' by adding the following line in 'autoexec.bat':

```
doskey shuffle=perl c:\perl\bin\shuffle.pl $*
```

References

GPL. (1991). *GNU general public license.* (Version 2. Published by the Free Software Fondation, `http://www.fsf.org`)

Luce, R. D. (1986). *Response times: Their role in inferring elementary mental organization.* New York: Oxford University Press.

Schwartz, R. L., & Phoenix, T. (2001). *Learning Perl.* O'Reilly.

Walsh, N. (1999). *Learning Perl/Tk.* O'Reilly.