# Restricted permutations; using the permute Package

**Gavin L. Simpson**
Environmental Change Research Centre — UCL

**Abstract**

*Keywords*: permutations, restricted permutations, time series, transects, spatial grids, split-plot designs, Monte Carlo resampling, R.

## 1. Introduction

In classical frequentist statistics, the significance of a relationship or model is determined by reference to a null distribution for the test statistic. This distribution is derived mathematically and the probability of achieving a test statistic as large or larger if the null hypothesis were true is looked-up from this null distribution. In deriving this probability, some assumptions about the data or the errors are made. If these assumptions are violated, then the validity of the derived $p$-value may be questioned.

An alternative to deriving the null distribution from theory is to generate a null distribution for the test statistic by randomly shuffling the data in some manner, refitting the model and deriving values for the test statistic for the permuted data. The level of significance of the test can be computed as the proportion of values of the test statistic from the null distribution that are equal to or larger than the observed value.

In many data sets, simply shuffling the data at random is inappropriate; under the null hypothesis, that data are not freely exchangeable. If there is temporal or spatial correlation, or the samples are clustered in some way, such as multiple samples collected from each of a number of fields. The **permute** was designed to provide facilities for generating these restricted permutations for use in randomisation tests.

## 2. Simple randomisation

As an illustration of both randomisation and the use of the **permute** package we consider a small data set of mandible length measurements on specimens of the golden jackal (*Canis aureus*) from the British Museum of Natural History, London, UK. These data were collected as part of a study comparing prehistoric and modern canids (Higham *et al.* 1980), and were analysed by Manly (2007). There are ten measurements of mandible length on both male and female specimens. The data are available in the `jackal` data frame supplied with **permute**.

```
R> require(permute)
R> data(jackal)
R> jackal

   Length   Sex
1     120   Male
```

```
2     107   Male
3     110   Male
4     116   Male
5     114   Male
6     111   Male
7     113   Male
8     117   Male
9     114   Male
10    112   Male
11    110 Female
12    111 Female
13    107 Female
14    108 Female
15    110 Female
16    105 Female
17    107 Female
18    106 Female
19    111 Female
20    111 Female
```

The interest is whether there is a difference in the mean mandible length between male and female golden jackals. The null hypothesis is that there is zero difference in mandible length between the two sexes or that females have larger mandible. The alternative hypothesis is that males have larger mandibles. The usual statistical test of this hypothesis is a one-sided $t$ test, which can be applied using `t.test()`

```
R> jack.t <-t.test(Length ~ Sex, data = jackal, var.equal = TRUE, alternative = "greater")
R> jack.t

        Two Sample t-test

data:  Length by Sex
t = 3.4843, df = 18, p-value = 0.001324
alternative hypothesis: true difference in means is greater than 0
95 percent confidence interval:
 2.411156      Inf
sample estimates:
  mean in group Male mean in group Female
               113.4                108.6
```

The observed $t$ is 3.484 with 18 df. The probability of observing a value this large or larger if the null hypothesis were true is 0.0013. Several assumptions have been made in deriving this $p$-value, namely

1. random sampling of individuals from the populations of interest,
2. equal population standard deviations for males and females, and
3. that the mandible lengths are normally distributed within the sexes.

Assumption 1 is unlikely to be valid for museum specimens such as these, that have been collected in some unknown manner. Assumption 2 may be valid, Fisher's $F$-test and a Fligner-Killeen test both suggest that the standard deviations of the two populations do not differ significantly

```
R> var.test(Length ~ Sex, data = jackal)
```

```
        F test to compare two variances

data:  Length by Sex
F = 2.681, num df = 9, denom df = 9, p-value = 0.1579
alternative hypothesis: true ratio of variances is not equal to 1
95 percent confidence interval:
  0.665931 10.793829
sample estimates:
ratio of variances
          2.681034
```

```
R> fligner.test(Length ~ Sex, data = jackal)

        Fligner-Killeen test of homogeneity of variances

data:  Length by Sex
Fligner-Killeen:med chi-squared = 0.7808, df = 1, p-value = 0.3769
```

This assumption may be relaxed using `var.equal = FALSE` (the default) in our call to `t.test()`, to employ Welch's modification for unequal variances. Assumption 3 may be valid, but with such a small sample we are able to reliably test this.

A randomisation test of the same hypothesis can be performed by randomly allocating ten of the mandible lengths to the male group and the remaining lengths to the female group. This randomisation is justified under the null hypothesis because the observed difference in mean mandible length between the two sexes is just a typical value for the difference in a sample if there were no difference in the population. An appropriate test statistic needs to be selected. We could use the $t$ statistic as derived in the $t$-test. Alternatively, we could base our randomisation test on the difference of means $D_i$ (male - female).

The main function in **permute** for providing random permutations is `shuffle()`. We can write our own randomisation test for the `jackal` data by first creating a function to compute the difference of means for two groups

```
R> meanDif <- function(x, grp) {
+  mean(x[grp == "Male"]) - mean(x[grp == "Female"])
+ }
```

which can be used in a simple `for()` loop to generate the null distribution for the difference of means. First, we allocate some storage to hold the null difference of means; here we use 4999 random permutations so allocate a vector of length 5000. Then we iterate, randomly generating an ordering of the `Sex` vector and computing the difference means for that permutation.

```
R> Djackal <- numeric(length = 5000)
R> N <- nrow(jackal)
R> set.seed(42)
R> for(i in seq_len(length(Djackal) - 1)) {
+     perm <- shuffle(N)
+     Djackal[i] <- with(jackal, meanDif(Length, Sex[perm]))
+ }
R> Djackal[5000] <- with(jackal, meanDif(Length, Sex))
```

The observed difference of means was added to the null distribution, because under the null hypothesis the observed allocation of mandible lengths to male and female jackals is just one of the possible random allocations.

The null distribution of $D_i$ can be visualised using a histogram, as shown in Figure 1. The observed difference of means (4.8) is indicated by the red tick mark.

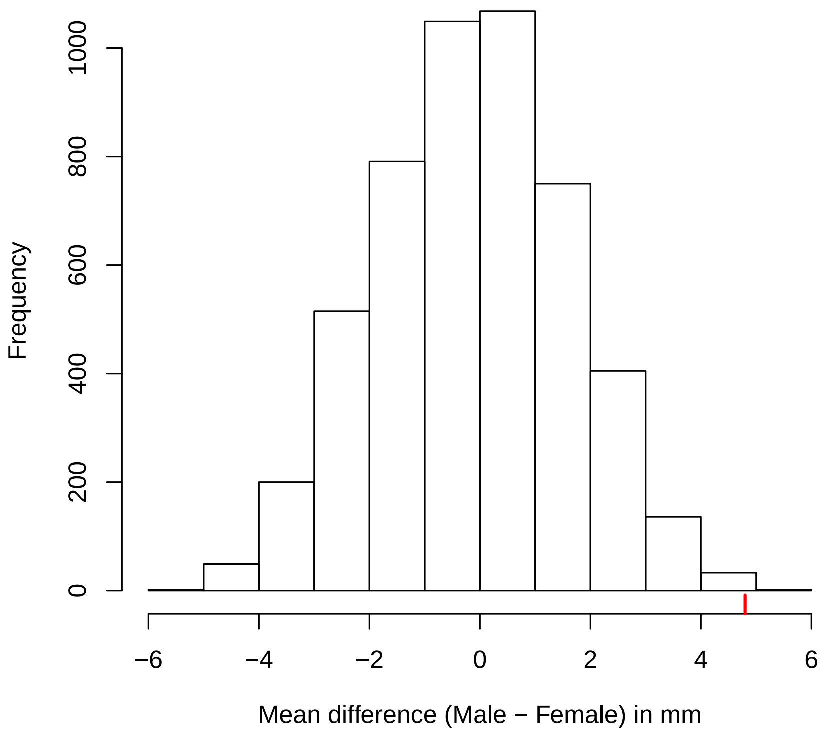

Figure 1: Distribution of the difference of mean mandible length in random allocations, ten to each sex.

```
R> hist(Djackal, main = "",
+      xlab = expression("Mean difference (Male - Female) in mm"))
R> rug(Djackal[5000], col = "red", lwd = 2)
```

The number of values in the randomisation distribution equal to or larger than the observed difference is

```
R> (Dbig <- sum(Djackal >= Djackal[5000]))
```

```
[1] 12
```

giving a permutational $p$-value of

```
R> Dbig / length(Djackal)
```

```
[1] 0.0024
```

which is comparable with that determined from the frequentist $t$-test, and indicate strong evidence against the null hypothesis of no difference.

In total there ${}^{20}C_{10} = 184,756$ possible allocations of the 20 observations to two groups of ten

```
R> choose(20, 10)
```

```
[1] 184756
```

so we have only evaluated a small proportion of these in the randomisation test.

The main workhorse function we used above was `shuffle()`. In this example, we could have used the base R function `sample()` to generate the randomised indices `perm` that were used to permute the `Sex` factor. Where `shuffle()` comes into it's own is for generating permutation indices from restricted permutation designs.

## 3. The shuffle() and shuffleSet() functions

In the previous section I introduced the `shuffle()` function to generate permutation indices for use in a randomisation test. Now we will take a closer look at `shuffle()` and explore the various restricted permutation designs from which it can generate permutation indices.

`shuffle()` has two arguments: i) `n`, the number of observations in the data set to be permuted, and ii) `control`, a list that defines the permutation design describing how the samples should be permuted.

```
R> args(shuffle)

function (n, control = permControl())
NULL
```

A series of convenience functions are provided that allow the user to set-up even quite complex permutation designs with little effort. The user only needs to specify the aspects of the design they require and the convenience functions ensure all configuration choices are set and passed on to `shuffle()`. The main convenience function is `permControl()`, which return a list specifying all the options available for controlling the sorts of permutations returned by `shuffle()`

```
R> str(permControl())

List of 10
 $ strata     : NULL
 $ nperm      : num 199
 $ complete   : logi FALSE
 $ within     :List of 5
  ..$ type    : chr "free"
  ..$ constant: logi FALSE
  ..$ mirror  : logi FALSE
  ..$ ncol    : NULL
  ..$ nrow    : NULL
 $ blocks     :List of 4
  ..$ type  : chr "none"
  ..$ mirror: logi FALSE
  ..$ ncol  : NULL
  ..$ nrow  : NULL
 $ maxperm    : num 9999
 $ minperm    : num 99
 $ all.perms  : NULL
 $ observed   : logi FALSE
 $ name.strata: chr "NULL"
 - attr(*, "class")= chr "permControl"
```

The defaults describe a random permutation design where all objects are freely exchangeable. Using these defaults, `shuffle(10)` amounts to `sample(1:10, 10, replace = FALSE)`:

```
R> set.seed(2)
R> (r1 <- shuffle(10))

 [1]  2  7  5 10  6  8  1  3  4  9

R> set.seed(2)
R> (r2 <- sample(1:10, 10, replace = FALSE))

 [1]  2  7  5 10  6  8  1  3  4  9

R> all.equal(r1, r2)

[1] TRUE
```

## 3.1. Generating restricted permutations

Several types of permutation are available in **permute**:

- Free permutation of objects
- Time series or line transect designs, where the temporal or spatial ordering is preserved.
- Spatial grid designs, where the spatial ordering is preserved in both coordinate directions
- Permutation of blocks or groups of samples.

The first three of these can be nested within the levels of a factor or to the levels of that factor, or to both. Such flexibility allows the analysis of split-plot designs using permutation tests.

`permControl()` is used to set up the design from which `shuffle()` will draw a permutation. `permControl()` has two main arguments that specify how samples are permuted *within* blocks of samples or at the block level itself. These are `within` and `blocks`. Two convenience functions, `Within()` and `Blocks()` can be used to set the various options for permutation.

For example, to permute the observations `1:10` assuming a time series design for the entire set of observations, the following control object would be used

```
R> set.seed(4)
R> x <- 1:10
R> CTRL <- permControl(within = Within(type = "series"))
R> perm <- shuffle(10, control = CTRL)
R> perm

 [1]  7  8  9 10  1  2  3  4  5  6

R> x[perm] ## equivalent

 [1]  7  8  9 10  1  2  3  4  5  6
```

It is assumed that the observations are in temporal or transect order. We only specified the type of permutation within blocks, the remaining options were set to their defaults via `Within()`.

A more complex design, with three blocks, and a 3 by 3 spatial grid arrangement within each block can be created as follows

```
R> set.seed(4)
R> block <- gl(3, 9)
R> CTRL <- permControl(strata = block,
+                      within = Within(type = "grid", ncol = 3, nrow = 3))
R> perm <- shuffle(length(block), control = CTRL)
R> perm

 [1]  6  4  5  9  7  8  3  1  2 14 15 13 17 18 16 11 12 10 22 23 24 25 26 27 19
[26] 20 21
```

Visualising the permutation as the 3 matrices may help illustrate how the data have been shuffled

```
R> ## Original
R> lapply(split(1:27, block), matrix, ncol = 3)

$`1`
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

$`2`
     [,1] [,2] [,3]
[1,]   10   13   16
[2,]   11   14   17
[3,]   12   15   18

$`3`
     [,1] [,2] [,3]
[1,]   19   22   25
[2,]   20   23   26
[3,]   21   24   27

R> ## Shuffled
R> lapply(split(perm, block), matrix, ncol = 3)

$`1`
     [,1] [,2] [,3]
[1,]    6    9    3
[2,]    4    7    1
[3,]    5    8    2

$`2`
     [,1] [,2] [,3]
[1,]   14   17   11
[2,]   15   18   12
[3,]   13   16   10

$`3`
     [,1] [,2] [,3]
[1,]   22   25   19
[2,]   23   26   20
[3,]   24   27   21
```

In the first grid, the lower-left corner of the grid was set to row 2 and column 2 of the original, to row 1 and column 2 in the second grid, and to row 3 column 2 in the third grid.

To have the same permutation within each level of `block`, use the `constant` argument of the `Within()` function, setting it to `TRUE`

```
R> set.seed(4)
R> CTRL <- permControl(strata = block,
+                      within = Within(type = "grid", ncol = 3, nrow = 3,
+                                      constant = TRUE))
R> perm2 <- shuffle(length(block), control = CTRL)
R> lapply(split(perm2, block), matrix, ncol = 3)

$`1`
     [,1] [,2] [,3]
[1,]    6    9    3
[2,]    4    7    1
[3,]    5    8    2

$`2`
     [,1] [,2] [,3]
[1,]   15   18   12
[2,]   13   16   10
[3,]   14   17   11

$`3`
     [,1] [,2] [,3]
[1,]   24   27   21
[2,]   22   25   19
[3,]   23   26   20
```

## 3.2. Generating sets of permutations with shuffleSet()

There are several reasons why one might wish to generate a set of $n$ permutations instead of repeatedly generating permutations one at a time. Interpreting the permutation design happens each time `shuffle()` is called. This is an unnecessary computational burden, especially if you want to perform tests with large numbers of permutations. Furthermore, having the set of permutations available allows for expedited use with other functions, they can be iterated over using `for` loops or the `apply` family of functions, and the set of permutations can be exported for use outside of R.

The `shuffleSet()` function allows the generation of sets of permutations from any of the designs available in **permute**. `shuffleSet()` takes an additional argument to that of `shuffle()`, `nset`, which is the number of permutations required for the set. Internally, `shuffle()` and `shuffleSet()` are very similar, with the major difference being that `shuffleSet()` arranges repeated calls to the workhorse permutation-generating functions with only the overhead associated with interpreting the permutation design once. `shuffleSet()` returns a matrix where the rows represent different permutations in the set.

As an illustration, consider again the simple time series example from earlier. Here I generate a set of 5 permutations from the design, with the results returned as a matrix

```
R> set.seed(4)
R> CTRL <- permControl(within = Within(type = "series"))
R> pset <- shuffleSet(10, nset = 5, control = CTRL)
R> pset
```

```
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    7    8    9   10    1    2    3    4    5     6
[2,]    2    3    4    5    6    7    8    9   10     1
[3,]    4    5    6    7    8    9   10    1    2     3
[4,]    4    5    6    7    8    9   10    1    2     3
[5,]   10    1    2    3    4    5    6    7    8     9
```

## Computational details

```
R> toLatex(sessionInfo())
```

- R version 2.15.0 Patched (2012-03-30 r58877), `x86_64-unknown-linux-gnu`
- Locale: `LC_CTYPE=en_GB.utf8`, `LC_NUMERIC=C`, `LC_TIME=en_GB.utf8`, `LC_COLLATE=C`, `LC_MONETARY=en_GB.utf8`, `LC_MESSAGES=en_GB.utf8`, `LC_PAPER=C`, `LC_NAME=C`, `LC_ADDRESS=C`, `LC_TELEPHONE=C`, `LC_MEASUREMENT=en_GB.utf8`, `LC_IDENTIFICATION=C`
- Base packages: base, datasets, grDevices, graphics, methods, stats, utils
- Other packages: permute 0.7-0
- Loaded via a namespace (and not attached): tools 2.15.0

## References

Higham C, Kijngam A, Manly B (1980). "An analysis of prehistoric canid remains from Thailand." *Journal of Archaeological Science*, **7**, 149–165.

Manly B (2007). *Randomization, bootstrap and Monte Carlo methods in biology.* 3rd edition. Chapman & Hall/CRC, Boca Raton.