Randomized layouts and sample size computations using dae in R

This document describes the randomization of simple experimental designs in R using the following functions:

`factor(x)`: creates a factor.

`rep(x, times = y)`: replicates each value of `x` the number of times specified by the corresponding element of `y`; `x` and `y` must be of equal length.

`rep(x, each = r, times = n)`: each value of `x` is replicated `r` times and then the whole of the result of that is repeated n times where n is a scalar.

`fac.divide(combined.factor, factor.names, order="standard")`: a nonstandard function that takes a `combined.factor` and divides into the factors specified in the list `factor.names`.

`fac.gen (generate, each=1, times=1, order="standard")`: generates the levels values of a set of factors, storing them in a data frame.

`fac.layout (unrandomized, nested.factors=NULL, randomized, seed=NULL)`: generates the randomized layout for an experiment by randomizing the randomized factors to the unrandomized factor taking into account the nesting amongst the unrandomized factors.

The functions `fac.divide`, `fac.gen` and `fac.layout` are in the package `dae` available from cran: http://http://cran.r-project.org/.

1. Completely randomized design

The following general expressions are used to obtain a randomized layout for a completely randomized design in R. You will need to a) assign a value to *n*, *t*, and *r* b) to replace the italicized names with ones for your particular experiment and c) choose a value for *s* to set the seed.

```
#
# Obtaining randomized layout for a CRD
#
CRD.unit <- list(Unit = n)
Treat <- factor(rep(1:t, times = r), labels=c("A","B",…))
CRD.lay <- fac.layout(unrandomized=CRD.unit, randomized=Treat,
                      seed=s)
CRD.lay
#remove Treat object in workspace to avoid using it by mistake
remove(Treat)
```

Example: a rat experiment

For example, consider an experiment in which the effects of three diets on rats is to be investigated. Suppose I have 6 rats to be fed one of three diets, 3 rats to be fed diet A, 2 diet B and 1 diet C.

The following output shows the use of the R function `fac.layout` from the `dae` package to produce the randomized layout for this example. The `unrandomized` argument gives the single unrandomized factor indexing the units in the experiment. The `randomized` argument specifies the factor, Diets, that is to be randomized. The `seed` argument is used so that the same randomized layout for a particular experiment can be generated at a later date. In general a different random small integer value, say between 0 and 1023, should be supplied as for this argument each time a new experimental layout is being obtained.

```
> #
> # Obtaining randomized layout for a CRD
> #
> n <- 6
> CRDRat.unit <- list(Rat = n)
> Diet <- factor(rep(c("A","B","C"), times = c(3,2,1)))
> CRDRat.lay <- fac.layout(unrandomized=CRDRat.unit, randomized=Diet, seed=695)
> CRDRat.lay
  Units Permutation Rat Diet
1     1           4   1    A
2     2           1   2    C
3     3           5   3    B
4     4           3   4    A
5     5           6   5    A
6     6           2   6    B
> #remove Diet object in workspace to avoid using it by mistake
> remove(Diet)
```

The process starts with the unrandomized and randomized factors in a systematic layout as follows:

```
> data.frame(fac.gen(CRDRat.unit), Diet)
  Rat Diet
1   1    A
2   2    A
3   3    A
4   4    B
5   5    B
6   6    C
```

Next a permutation of the Rats is selected. In our case there are 6! Possible permutations and the one chosen is shown in the Rat column of the following output:

```
> CRDRat.lay[CRDRat.lay$Permutation,]
  Units Permutation Rat Diet
4     4           3   4    A
1     1           4   1    A
5     5           6   5    A
3     3           5   3    B
6     6           2   6    B
2     2           1   2    C
```

Applying this permutation leads to the randomization given above.

■

2. Randomized complete block design

The following general expressions are used to obtain a randomized layout for an RCBD in R, the layout being stored in the data frame `RCBD.lay`. To use these expressions to generate a layout for a particular case, you will need to assign values to *b*, *t* and *n* prior to using them and to substitute the actual names for *Blocks*, *Units, Treats* and the design object to contain them. Also, the labels for the treatments are optional. The crucial feature, that makes this design different from a completely randomized design, is that there are two unrandomized factors indexing the units and there is nesting between these factors: *Units* are nested within *Blocks*. This is because our aim is to randomize the treatments to the *Units* **within** each *Block*. The `nested.factors` argument to `fac.layout` is used to specify this.

```
RCBD.unit <- list(Blocks = b, Units = t)
RCBD.nest <- list(Units = "Blocks")
Treat <- factor(rep(1:t, times = b), labels=c("A","B",…))
data.frame(fac.gen(RCBD.unit), Treat) #basic systematic layout
RCBD.lay <- fac.layout(unrandomized = RCBD.unit,
                       nested.factors = RCBD.nest,
                       randomized = Treat, seed = s)
RCBD.lay
RCBD.lay[RCBD.lay$Permutation,]
```

These expressions will output three sets of factor values: a) the systematic layout on which the randomization is based; b) the randomized layout; c) the permutation of the unrandomized factors that was selected for the randomization and used to get from the systematic to the randomized layout.

Example: Penicillin yield

In this example, (Box, Hunter and Hunter, 2005, section 4.2) the effects of four treatments (A, B, C and D) on the yield of penicillin are to be investigated. It was known that corn steep liquor, an important raw material in producing penicillin, is highly variable from one blending of it to another. So, to ensure that the results of the experiment apply to more than one blend, several blends are to be used in the experiment. Thus the trial was conducted using the same blend in four flasks and randomizing the four treatments to these four flasks. Altogether five blends were utilized.

The names to be used for the blocks, units and treatments for this example are Blends, Flask and Treat, respectively. Also, $b = 5$ and $t = 4$ so that $n = 20$. Assigning these values and substituting these names into the expressions above, yields the following output for this case:

```
> b <- 5
> t <- 4
> n <- b*t
> RCBDPen.unit <- list(Blend=b, Flask=t)
> RCBDPen.nest <- list(Flask = "Blend")
> Treat <- factor(rep(1:t, times=b), labels=c("A","B","C","D"))
> data.frame(fac.gen(RCBDPen.unit), Treat) #basic systematic layout
   Blend Flask Treat
1      1     1     A
2      1     2     B
3      1     3     C
4      1     4     D
5      2     1     A
6      2     2     B
7      2     3     C
8      2     4     D
9      3     1     A
10     3     2     B
11     3     3     C
12     3     4     D
13     4     1     A
14     4     2     B
15     4     3     C
16     4     4     D
17     5     1     A
18     5     2     B
19     5     3     C
20     5     4     D
> RCBDPen.lay <- fac.layout(unrandomized = RCBDPen.unit,
+                           nested.factors = RCBDPen.nest,
+                           randomized = Treat, seed = 311)
> RCBDPen.lay
   Units Permutation Blend Flask Treat
1      1          11     1     1     C
2      2          12     1     2     B
3      3          10     1     3     D
4      4           9     1     4     A
5      5          13     2     1     C
6      6          15     2     2     D
7      7          16     2     3     B
8      8          14     2     4     A
9      9           8     3     1     D
10    10           7     3     2     C
11    11           5     3     3     A
12    12           6     3     4     B
13    13          17     4     1     A
```

```
14    14          19     4     2     D
15    15          20     4     3     B
16    16          18     4     4     C
17    17           4     5     1     A
18    18           2     5     2     D
19    19           1     5     3     B
20    20           3     5     4     C
> RCBDPen.lay[RCBDPen.lay$Permutation,]
   Units Permutation Blend Flask Treat
11    11           5     3     3     A
12    12           6     3     4     B
10    10           7     3     2     C
9      9           8     3     1     D
13    13          17     4     1     A
15    15          20     4     3     B
16    16          18     4     4     C
14    14          19     4     2     D
8      8          14     2     4     A
7      7          16     2     3     B
5      5          13     2     1     C
6      6          15     2     2     D
17    17           4     5     1     A
19    19           1     5     3     B
20    20           3     5     4     C
18    18           2     5     2     D
4      4           9     1     4     A
2      2          12     1     2     B
1      1          11     1     1     C
3      3          10     1     3     D
```

So with the first blend, the Treatments are to be done in the order C, B, D, A. ■

3. Generalized randomized complete block design

The following general expressions are used to obtain a randomized layout for a generalized RCBD in R — they are similar to those for the RCBD in the previous section. To use these expressions to generate a layout for a particular case, you will need to assign values to *b*, *t*, *g*, *u* and *n* prior to using them and to substitute the actual names for *Blocks*, *Units, Treats* and the design object to contain them. Also, the labels for the treatments are optional.

```
GRCBD.unit <- list(Blocks=b, Units=u)
GRCBD.nest <- list(Units="Blocks")
Treat <- factor(rep(1:t, times=b*g),
      labels=c("A","B","C","D"))
data.frame(fac.gen(GRCBD.unit), Treat)
GRCBD.lay <- fac.layout(unrandomized=GRCBD.unit,
                        nested.factors=GRCBD.nest,
                        randomized=Treat, seed=s)
GRCBD.lay
```

Example: Design for a wheat experiment

```
> b <- 2
> t <- 4
> g <- 3
> u <- t*g
> n <- b*u
> GRCBDWheat.unit <- list(Blocks=b, Plots=u)
> GRCBDWheat.nest <- list(Plots="Blocks")
> Treat <- factor(rep(1:t, times=b*g), labels=c("A","B","C","D"))
> data.frame(fac.gen(GRCBDWheat.unit), Treat)
   Blocks Plots Treat
1       1     1     A
2       1     2     B
3       1     3     C
4       1     4     D
5       1     5     A
6       1     6     B
7       1     7     C
8       1     8     D
9       1     9     A
10      1    10     B
11      1    11     C
12      1    12     D
13      2     1     A
14      2     2     B
15      2     3     C
16      2     4     D
17      2     5     A
18      2     6     B
19      2     7     C
20      2     8     D
21      2     9     A
22      2    10     B
23      2    11     C
24      2    12     D
> GRCBDWheat.lay <- fac.layout(unrandomized=GRCBDWheat.unit,
+                               nested.factors=GRCBDWheat.nest,
+                               randomized=Treat, seed=399)
> GRCBDWheat.lay
   Units Permutation Blocks Plots Treat
1      1          20      1     1     C
2      2          22      1     2     D
3      3          24      1     3     D
4      4          18      1     4     C
5      5          17      1     5     B
6      6          21      1     6     B
7      7          23      1     7     A
8      8          13      1     8     A
9      9          14      1     9     D
10    10          19      1    10     A
11    11          16      1    11     B
12    12          15      1    12     C
13    13           7      2     1     D
14    14           5      2     2     A
15    15           1      2     3     D
16    16           3      2     4     C
17    17           8      2     5     A
18    18           6      2     6     D
19    19          12      2     7     B
20    20           2      2     8     A
21    21          10      2     9     B
22    22          11      2    10     B
23    23           4      2    11     C
24    24           9      2    12     C
```

4. Latin square design

The general set of expressions for obtaining a Latin Square layout are:

```
t <- 4
n <- t*t
LS.unit <- list(Rows=t, Columns=t)
Treats <- factor(c(the unrandomized layout for the Latin Square),
       labels=c("A","B" …))
LS.lay <- fac.layout(unrandomized=LS.unit, randomized=Treats,
       seed=s)
remove("Treats")
LS.lay
```

To use this set of expressions to generate a layout for a particular case, you will need to assign the actual values for *t* and *n* and substitute the actual names for *Rows*, *Columns, Treats* and the design object to contain them. Also, you will need to put in an unrandomized Latin square layout and, optionally, the labels for the treatments. Note that, like the randomized complete block design the Latin square involves two unrandomized factors, *Rows* and *Columns* say, that index the units. However, for the Latin square, as the *Rows* and *Columns* are to be randomized independently, they are not nested (they are crossed), and so the `nested.factors` argument is not set and so will be `NULL`.

Example: Pollution effects of petrol additives

Suppose that four cars and four drivers are employed in a study of possible differences between four petrol additives as far as their effect on pollution is concerned (Box, Hunter and Hunter, 2005, section 4.4). Even if the cars are identical models, consistent differences are likely to occur between individual cars. Even though each driver may do their best to drive in the manner required by the test, consistent differences are likely to occur between the drivers. It would be desirable to isolate both car-to-car and driver-to-driver differences. A 4×4 Latin square would enable this to be done.

The names to be used for the rows, columns and treats for this example are Cars, Drives and Additives, respectively. Also, $t = 4$ and a suitable design was obtained from Box, Hunter and Hunter. Substituting these into the expressions above, yields the following expressions to be used in this case:

```
> t <- 4
> n <- t*t
> LSPolut.unit <- list(Drivers=t, Cars=t)
> Additives <- factor(c(1,2,3,4, 4,3,2,1, 2,1,4,3, 3,4,1,2),
+                                                  labels=c("A","B","C","D"))
> LSPolut.lay <- fac.layout(unrandomized=LSPolut.unit, randomized=Additives,
+                           seed=941)
> remove("Additives")
```

```
> LSPolut.lay
   Units Permutation Drivers Cars Additives
1      1          11       1    1         B
2      2          12       1    2         A
3      3          10       1    3         C
4      4           9       1    4         D
5      5           7       2    1         A
6      6           8       2    2         B
7      7           6       2    3         D
8      8           5       2    4         C
9      9          15       3    1         D
10    10          16       3    2         C
11    11          14       3    3         A
12    12          13       3    4         B
13    13           3       4    1         C
14    14           4       4    2         D
15    15           2       4    3         B
16    16           1       4    4         A
```

Thus the randomized layout is:

4 x 4 Latin square

		Car			
		1	2	3	4
	I	B	A	C	D
	II	A	B	D	C
Drivers	III	D	C	A	B
	IV	C	D	B	A

(Additives A, B, C, D)

■

5. Sets of Latin Squares

Examples: Pollution effects of petrol additives (continued)

In the above example of a single Latin square, *Pollution effects of petrol additives*, four cars and four drivers are employed in a study of possible differences between four petrol additives as far as their effect on pollution is concerned. To isolate both car-to-car and driver-to-driver differences a 4×4 Latin square was employed. However, this allowed only 6 degrees of freedom for the Residual sum of squares. To overcome this problem sets of Latin squares were considered in which the Latin square was repeated using:

1. using the same drivers (rows) and cars (columns) in each set;
2. using new drivers (rows) but the same cars (columns);
2b. using the same drivers (rows) but new cars (columns); or
3. using new cars (rows) and drivers (columns).

The unrandomized factors that index the units for all cases of sets of Latin squares are the same as is the randomized factor. The only thing that varies is the nesting between the unrandomized factors and so it is only the `nested.factor` argument that varies between cases. The common R expressions are:

```
r <- 2
t <- 4
n <- r*t*t
LSRepeat.unit <- list(Occasion=r, Drivers=t, Cars=t)
Additives <- factor(rep(c(1,2,3,4, 4,3,2,1, 2,1,4,3, 3,4,1,2),
                        times=r), labels=c("A","B","C","D"))
```

Having run these expressions in R, the additional R expressions required for each case are as follows:

```
#
# Sets of Latin squares - case 1
#
LSRepeat1.lay <- fac.layout(unrandomized=LSRepeat.unit,
                            randomized=Additives, seed=914)
LSRepeat1.lay
#
# Sets of Latin squares - case 2
#
LSRepeat2.nest <- list(Drivers="Occasion")
LSRepeat2.lay <- fac.layout(unrandomized=LSRepeat.unit,
                            nested.factors=LSRepeat2.nest,
                            randomized=Additives, seed=149)
LSRepeat2.lay
#
# Sets of Latin squares - case 2b
#
LSRepeat2b.nest <- list(Cars="Occasion")
LSRepeat2b.lay <- fac.layout(unrandomized=LSRepeat.unit,
                             nested.factors=LSRepeat2b.nest,
                             randomized=Additives, seed=194)
LSRepeat2b.lay
#
# Sets of Latin squares - case 3
#
LSRepeat3.nest <- list(Cars="Occasion", Drivers="Occasion")
LSRepeat3.lay <- fac.layout(unrandomized=LSRepeat.unit,
                            nested.factors=LSRepeat3.nest,
                            randomized=Additives, seed=419)
LSRepeat3.lay
```

The layouts these expressions produce are given in chapter 3, *Sets of Latin squares*. The randomized layout for a set consisting of any number of *r* squares can be obtained by setting *r* to the number of squares desired. In these examples, where $r = 2$, there are 2 copies. Also, one could employ different starting squares for each square in the set in case 3 by suitable defining the factor `Additives`. Of course, for a different experiment, the names of the factors and the starting square would need to be changed with Occasions corresponding to the sets, Drivers the rows and Cars the columns. Also, because of the order of the factors in `LSRepeat.unit`, the layout will be listed in standard order for Sets then Rows then Columns. However, this can be changed by varying the order of the factors in `LSRepeat.unit`.

6. Factorial experiments

Layouts for factorial experiments can be obtained in R using the expressions for the chosen design when only a single-factor is involved. The difference with factorial experiments is that the several treatment factors need to be entered. Their values can be generated using the `fac.gen` function. It is likely to be necessary to use either the `each` or `times` arguments to generate the replicate combinations.

Example: Fertilizing oranges

Suppose an experimenter is interested in investigating the effect of nitrogen and phosphorus fertilizer on yield of oranges. It was decided to investigate 3 levels of Nitrogen (viz 0,30,60 kg/ha) and 2 levels of Phosphorus (viz. 0,20 kg/ha). The yield after six months was measured.

For a factorial experiment, the treatments are all possible combinations of the 3 Nitrogen $\times$ 2 Phosphorus levels: $3 \times 2 = 6$ treatments. The treatment combinations, in standard order, are:

Treatment	N	P
1	0	0
2	0	20
3	30	0
4	30	20
5	60	0
6	60	20

A layout for this experiment in a CRD with three replicates of each treatment is generated in R as shown in the following output.

```
> #
> # CRD
> #
> n <- 18
> CRDFac2.unit <- list(Seedling = n)
> CRDFac2.ran <- fac.gen(list(N = c(0, 30, 60), P = c(0, 20)), times = 3)
> CRDFac2.lay <- fac.layout(unrandomized = CRDFac2.unit,
+                           randomized = CRDFac2.ran, seed = 105)
> remove("CRDFac2.unit", "CRDFac2.ran")
> CRDFac2.lay
   Units Permutation Seedling  N  P
1      1           2        1 30 20
2      2          18        2  0  0
3      3           4        3 30  0
4      4           5        4 30  0
5      5           7        5 30 20
6      6          12        6 30  0
7      7          15        7 60  0
8      8          13        8  0  0
9      9           6        9 60  0
10    10           1       10 60  0
11    11          10       11 30 20
12    12          16       12 60 20
13    13           8       13  0 20
14    14          14       14  0 20
15    15           3       15  0  0
16    16          11       16 60 20
```

```
17    17           9       17 60 20
18    18          17       18  0 20
```

Note the assignment of the treatment factor names and associated levels to a list Note the assignment of the generation of treatment values using `fac.gen` that creates 3 copies of the levels combinations of the two factors N and P, that have 3 and 2 levels respectively, and stores these in the `data.frame` `CRDFac2.ran`.

Again, R can be used to obtain the layout as shown in the following output:

```
> #
> # RCBD
> #
> b <- 3
> t <- 6
> n <- b*t
> RCBDFac2.unit <- list(Blocks=b, Seedling=t)
> RCBDFac2.nest <- list(Seedling = "Blocks")
> RCBDFac2.ran <- fac.gen(list(N = c(0, 30, 60), P = c(0, 20)), times = 3)
> RCBDFac2.lay <- fac.layout(unrandomized = RCBDFac2.unit,
+                            nested.factors = RCBDFac2.nest,
+                            randomized = RCBDFac2.ran, seed = 555)
> remove("RCBDFac2.ran", "RCBDFac2.unit", "RCBDFac2.nest")
> RCBDFac2.lay
   Units Permutation Blocks Seedling  N  P
1      1           2      1        1 30 20
2      2           6      1        2  0  0
3      3           5      1        3 60  0
4      4           1      1        4 60 20
5      5           3      1        5 30  0
6      6           4      1        6  0 20
7      7          16      2        1  0  0
8      8          13      2        2  0 20
9      9          17      2        3 60  0
10    10          14      2        4 30  0
11    11          15      2        5 60 20
12    12          18      2        6 30 20
13    13           7      3        1  0 20
14    14           8      3        2 30 20
15    15          10      3        3 60  0
16    16          12      3        4  0  0
17    17           9      3        5 30  0
18    18          11      3        6 60 20
```

Finally, the experiment could be laid out in a 6×6 Latin square. Once you have obtained a standard Latin square, R can be used to obtain the layout as shown in the following output.

```
> #
> # LS
> #
> t <- 6
> n <- t*t
> Treats <- c(1,2,3,4,5,6, 2,1,6,5,3,4, 3,6,2,1,4,5,
+                   4,3,5,2,6,1, 5,4,1,6,2,3, 6,5,4,3,1,2)
> LSFac2.ran <- fac.divide(Treats, list(N=c(0,30,60), P=c(0, 20)))
> LSFac2.unit <- list(Rows=t, Columns=t)
> LSFac2.lay <- fac.layout(unrandomized = LSFac2.unit,
+                          randomized = LSFac2.ran, seed = 559)
> remove("Treats", "LSFac2.unit", "LSFac2.ran")
```

```
> LSFac2.lay
   Units Permutation Rows Columns  N  P
1      1          29    1       1  0 20
2      2          27    1       2 30  0
3      3          30    1       3 60  0
4      4          26    1       4  0  0
5      5          28    1       5 60 20
6      6          25    1       6 30 20
7      7          35    2       1 60  0
8      8          33    2       2  0  0
9      9          36    2       3 60 20
10    10          32    2       4 30 20
11    11          34    2       5 30  0
12    12          31    2       6  0 20
13    13          11    3       1 30  0
14    14           9    3       2 60 20
15    15          12    3       3 30 20
16    16           8    3       4  0 20
17    17          10    3       5 60  0
18    18           7    3       6  0  0
19    19          23    4       1  0  0
20    20          21    4       2  0 20
21    21          24    4       3 30  0
22    22          20    4       4 60 20
23    23          22    4       5 30 20
24    24          19    4       6 60  0
25    25          17    5       1 60 20
26    26          15    5       2 30 20
27    27          18    5       3  0 20
28    28          14    5       4 60  0
29    29          16    5       5  0  0
30    30          13    5       6 30  0
31    31           5    6       1 30 20
32    32           3    6       2 60  0
33    33           6    6       3  0  0
34    34           2    6       4 30  0
35    35           4    6       5  0 20
36    36           1    6       6 60 20
```

Note the use of `fac.divide` to create the N and P factors from a factor for treatments numbered 1–6; `fac.gen` cannot be used here because the treatments are not in a patterned order. ■

7. Two-level factorial experiments

In these experiments we will generally use "-" and "+" for the two levels of each factor. These will be stored in the object `mp` (**m**inus-**p**lus) using the assignment

```
mp <- c("-", "+")
```

Also, a one-line function `mpone` that converts the first two levels of a factor in –1 and +1 respectively will be used where multiplication of the levels of factors needs to be performed. The function, available from the library *dae* is:

```
mpone <- function(factor){2 * as.numeric(factor) - 3}
```

a) Replicated two-level factorial experiments

In this example (Box, Hunter and Hunter, 2005, section 5.4), a 2^3 experiment with factors Te, C and K are to be replicated twice. The replicates of each treatment combinations are generated using `each = 2`.

```
> #
> # Replicated two-level factorial
> #
> n <- 16
> mp <- c("-", "+")
> Fac3.2Level.Rep.ran <- fac.gen(generate = list(Te = mp, C = mp, K = mp), each =
2,
+                           order="yates")
> Fac3.2Level.Rep.unit <- list(Runs = n)
> Fac3.2Level.Rep.lay <- fac.layout(unrandomized = Fac3.2Level.Rep.unit,
+                              randomized = Fac3.2Level.Rep.ran, seed = 625)
> Fac3.2Level.Rep.lay
   Units Permutation Tests Te C K
1      1           5     1  - - +
2      2           8     2  + - +
3      3          14     3  + + -
4      4           7     4  + + -
5      5           6     5  - - -
6      6          12     6  - + -
7      7           4     7  + - -
8      8           3     8  - - -
9      9          10     9  + + +
10    10           1    10  - - +
11    11           2    11  - + +
12    12          16    12  - + -
13    13          11    13  + + +
14    14          15    14  + - -
15    15          13    15  - + +
16    16           9    16  + - +
```

b) Unreplicated two-level factorial experiments

```
> #
> # Unreplicated two-level factorial
> #
> n <- 8
> mp <- c("-", "+")
> Fac3.2Level.Unrep.ran <- fac.gen(list(A = mp, B = mp, C = mp), order="yates")
> Fac3.2Level.Unrep.unit <- list(Runs = n)
> Fac3.2Level.Unrep.lay <- fac.layout(unrandomized = Fac3.2Level.Unrep.unit,
+                                     randomized = Fac3.2Level.Unrep.ran, seed=333)
> remove("Fac3.2Level.Unrep.ran")
> Fac3.2Level.Unrep.lay
  Units Permutation Runs A B C
1     1           4    1 - - +
2     2           2    2 + - -
3     3           8    3 + + +
4     4           5    4 - - -
5     5           1    5 + + -
6     6           7    6 - + +
7     7           6    7 + - +
8     8           3    8 - + -
```

c) Confounded two-level factorial experiments

To obtain layouts for the designs involving total confounding in R, use `fac.gen` to generate the combinations of the treatment factors to be observed, compute the

Block factors from the treatment factors and use instructions similar to the randomized complete block design to randomize the order of the blocks and the treatment combinations within blocks. Note the use of the one-line function `mpone` from the library *dae* to convert the first two levels of a factor into the numeric values –1 and +1.

Example: Complete sets of factorial treatments in 2 blocks

The output, including expressions, for producing a layout for this design is given below. Note that, in practice, one would need to choose a random number between 0 and 1023 to use for the `seed`— the alternative is to leave the argument out but then the layout cannot be reproduced.

```
> #
> # 3 factors in two blocks
> #
> # set up treatment factors
> #
> mp <- c("-", "+")
>  Fac3Conf.2Blocks.ran <- fac.gen(generate = list(A = mp, B = mp, C = mp),
order="yates")
> attach(Fac3Conf.2Blocks.ran)
> Blocks <- factor(mpone(A)*mpone(B)*mpone(C), labels=c("1", "2"))
> detach(Fac3Conf.2Blocks.ran)
> Fac3Conf.2Blocks.ran <- Fac3Conf.2Blocks.ran[order(Blocks),]
> remove("Blocks")
> #
> # randomize
> #
> b <- 2
> m <- 4
> n <- b*m
> Fac3Conf.2Blocks.nest <- list(Runs = "Blocks")
> Fac3Conf.2Blocks.unit <- list(Blocks=b, Runs=m)
> Fac3Conf.2Blocks.lay <- fac.layout(unrandomized = Fac3Conf.2Blocks.unit,
+                                    nested.factors = Fac3Conf.2Blocks.nest,
+                                    randomized = Fac3Conf.2Blocks.ran, seed = 395)
> Fac3Conf.2Blocks.lay
  Units Permutation Blocks Runs A B C
1     1           6      1    1 - - +
2     2           5      1    2 - + -
3     3           8      1    3 + - -
4     4           7      1    4 + + +
5     5           3      2    1 + + -
6     6           2      2    2 - - -
7     7           1      2    3 - + +
8     8           4      2    4 + - +
```

Example: Repeated two block experiment

The expressions for a repeated two-block experiment are similar to those for the unreplicated experiment — the major differences are the use of the `each` (or `times`) argument in `fac.gen` and the generation of an extra factor, Reps say, indexing the replications. The output, including expressions, for producing a layout for this design is given below. Note that, in practice, one would need to choose a random number between 0 and 1023 to use for the `seed`— the alternative is to leave the argument out but then the layout cannot be reproduced.

```
> #
> # repeats of 3 factors in two blocks
> #
> #
> # set up treatment factors
> #
> mp <- c("-", "+")
> Fac3Conf.2Blocks.Reps.ran <- fac.gen(generate = list(A = mp, B = mp, C = mp),
+                                      times = 2, order="yates")
> attach(Fac3Conf.2Blocks.Reps.ran)
> Reps = factor(rep(1:2, each=8))
> Blocks <- factor(mpone(A)*mpone(B)*mpone(C), labels=c("1", "2"))
> detach(Fac3Conf.2Blocks.Reps.ran)
> Fac3Conf.2Blocks.Reps.ran <- Fac3Conf.2Blocks.Reps.ran[order(Reps, Blocks),]
> remove("Reps","Blocks")
> #
> # randomize
> #
> set.seed(911)
> b <- 4
> m <- 4
> n <- b * m
> Fac3Conf.2Blocks.Reps.nest <- list(Runs = "Blocks")
> Fac3Conf.2Blocks.Reps.unit <- list(Blocks=b, Runs=m)
> Fac3Conf.2Blocks.Reps.lay <- fac.layout(
+                                  unrandomized = Fac3Conf.2Blocks.Reps.unit,
+                                  nested.factors = Fac3Conf.2Blocks.Reps.nest,
+                                  randomized = Fac3Conf.2Blocks.Reps.ran,
+                                  seed = 911)
> Fac3Conf.2Blocks.Reps.lay
   Units Permutation Blocks Runs A B C
1      1           7      1    1 - - +
2      2           8      1    2 - + -
3      3           6      1    3 + + +
4      4           5      1    4 + - -
5      5           4      2    1 - + +
6      6           2      2    2 + - +
7      7           1      2    3 - - -
8      8           3      2    4 + + -
9      9          15      3    1 + + +
10    10          14      3    2 - - +
11    11          13      3    3 + - -
12    12          16      3    4 - + -
13    13          11      4    1 + - +
14    14          12      4    2 + + -
15    15          10      4    3 - - -
16    16           9      4    4 - + +
```

Example: Repeated four block experiment

The output, including expressions, for producing a layout for this design is given below. Note that, in practice, one would need to choose a random number between 0 and 1023 to use for the `seed`— the alternative is to leave the argument out but then the layout cannot be reproduced.

```
> #
> # repeats of 3 factors in four blocks
> #
> #
> # set up treatment factors
> #
> Fac3Conf.4Blocks.Reps.ran <- fac.gen(generate = list(A = mp, B = mp, C = mp),
+                                      times = 2, order="yates")
> attach(Fac3Conf.4Blocks.Reps.ran)
```

```
> Reps = factor(rep(1:2, each=8))
> B1 <- factor(mpone(A)*mpone(B), labels=c("1", "2"))
> B2 <- factor(mpone(A)*mpone(C), labels=c("1", "2"))
> Blocks <- fac.combine(list(B1,B2))
> detach(Fac3Conf.4Blocks.Reps.ran)
> Fac3Conf.4Blocks.Reps.ran <- Fac3Conf.4Blocks.Reps.ran[order(Reps,Blocks),]
> remove("Reps","B1", "B2", "Blocks")
> #
> # randomize
> #
> set.seed(111)
> b <- 8
> m <- 2
> n <- b*m
> Fac3Conf.4Blocks.Reps.nest <- list(Runs = "Blocks")
> Fac3Conf.4Blocks.Reps.unit <- list(Blocks=b, Runs=m)
> Fac3Conf.4Blocks.Reps.lay <- fac.layout(
+                                    unrandomized = Fac3Conf.4Blocks.Reps.unit,
+                                    nested.factors = Fac3Conf.4Blocks.Reps.nest,
+                                    randomized = Fac3Conf.4Blocks.Reps.ran,
+                                    seed = 111)
> Fac3Conf.4Blocks.Reps.lay
   Units Permutation Blocks Runs A B C
1      1          14      1    1 + + -
2      2          13      1    2 - - +
3      3          15      2    1 - - +
4      4          16      2    2 + + -
5      5           4      3    1 + - -
6      6           3      3    2 - + +
7      7           9      4    1 - + -
8      8          10      4    2 + - +
9      9           5      5    1 - - -
10    10           6      5    2 + + +
11    11           7      6    1 + + +
12    12           8      6    2 - - -
13    13           1      7    1 - + +
14    14           2      7    2 + - -
15    15          12      8    1 - + -
16    16          11      8    2 + - +
```

d) Fractional two-level factorial experiments

The steps to be carried out to obtain the treatments for a 2^{k-p} fractional factorial design are:

1. Use `fac.gen` to generate the levels combinations of the first $k-p$ factors in a `design` because the design will be complete in this number of factors.
2. From the generators, such as can be obtained from the *Table of fractional factorial designs* given in section VIII.D, *Fractional factorial designs at two levels*, compute and add to the `design` the remaining factors —the one-line function `mpone` from the library *dae* is used to convert the first two levels of a factor into the numeric values –1 and +1.

If required, randomize the layout using the same instructions as for the unreplicated 2^k experiment.

Example: A bike experiment

The following expressions are used to generate a randomized layout for the first fraction of the experiment described in Box, Hunter and Hunter (2005, section 6.5). The layout is stored in a `data.frame` named `Frf7Bike.lay`:

```
mp <- c("-", "+")
fnames <- list(Seat = mp, Dynamo = mp, Handbars = mp)
Frf7Bike.ran <- fac.gen(generate = fnames, order = "yates")
attach(Frf7Bike.ran)
Frf7Bike.ran$Gear <- factor(mpone(Seat)*mpone(Dynamo), labels = mp)
Frf7Bike.ran$Raincoat <- factor(mpone(Seat)*mpone(Handbars), labels = mp)
Frf7Bike.ran$Brekkie <- factor(mpone(Dynamo)*mpone(Handbars), labels = mp)
Frf7Bike.ran$Tyres <- factor(mpone(Seat)*mpone(Dynamo)*mpone(Handbars),
                             labels = mp)
detach(Frf7Bike.ran)
Frf7Bike.unit <- list(Runs = 8)
Frf7Bike.lay <- fac.layout(unrandomized = Frf7Bike.unit,
                           randomized = Frf7Bike.ran)
Frf7Bike.lay
```

Note that `fac.gen` generates only the first 3 of the 7 factors as this design is complete in just these 3 factors. The remaining 4 factors are generated as combinations of these first 3 factors.

The following expressions generate a randomized layout for the second fraction (Box, Hunter and Hunter, 2005, section 6.8) in a `data.frame` named `Frf7Bike2.lay` and then combines the two fractions into a single `data.frame` named `Frf7Bike.Both.lay`:

```
Frf7Bike2.ran <- Frf7Bike.ran
attach(Frf7Bike2.ran)
Frf7Bike2.ran$Gear <- factor(-mpone(Seat)*mpone(Dynamo), labels = mp)
detach(Frf7Bike2.ran)
Frf7Bike2.lay <- fac.layout(unrandomized = Frf7Bike.unit,
                            randomized = Frf7Bike2.ran)
Frf7Bike.Both.lay <- rbind(Frf7Bike.lay,Frf7Bike2.lay)
Frf7Bike.Both.lay <- data.frame(Block = factor(rep(1:2, each=8)),
                                Frf7Bike.Both.lay)
Frf7Bike.Both.lay
```

8. Split-plot experiments

The general set of expressions for using R to obtain a layout for the standard split-plot experiment, with *r* blocks, *a* main-plot treatments and *b* subplot treatments, is as follows:

The general expressions for using R to obtain a layout for the standard split-plot experiment, with *r* blocks, *a* main-plot treatments and *b* sub-plot treatments, are given below. To use these expressions to generate a layout for a particular case, you will need to assign values to *r*, *a*, and *b* prior to using them and to substitute the actual names for *Blocks*, *MainPlots, SubPlots*, *A*, *B* and the *Experiment*. Also, the labels for the treatments are optional.

```
fnames <- list(A = a, B = b)
Experiment.ran <- fac.gen(generate = fnames, times = r)
Experiment.unit <- list(Blocks=r, MainPlots=a, SubPlots=b)
Experiment.nest <- list(MainPlots = "Blocks",
                        SubPlots = c("Blocks", "MainPlots"))
Experiment.lay <- fac.layout(unrandomized = Experiment.unit,
                             nested.factors = Experiment.nest,
                             randomized = Experiment.ran,
                             seed = 1025)
Experiment.lay
```

It is very important that the order in the `fac.gen` function has the main-plot treatment factor (A) before the subplot treatment factor (B). This is because `fac.gen` generates the levels combinations in standard order and so the first factor will correspond to the generation of *MainPlots* and the second factor to *SubPlots*.

Example: Production rate experiment

Johnson and Leone (1964, section 15.7) describe an experiment in which the researcher is interested in comparing the effects of 3 methods of work organization and 3 sources of raw material on the production rate of a certain product. It is decided that four factories are to be used in the experiment and that each factory is to be divided into three areas. The methods of work organization are to be assigned at random to areas. Each area is to be subdivided into 3 parts and the source of raw material for each part is obtained by randomizing the three sources to the three parts. The layout for this experiment can be obtained using the following R expressions:

```
> #
> # standard split-plot
> #
> r <- 4
> a <- 3
> b <- 3
> fnames <- list(Methods = c(1:a), Sources = c("A","B","C"))
> SPLProd.ran <- fac.gen(generate = fnames, times = r)
> SPLProd.unit <- list(Factories=r, Areas=c("I","II","III"), Parts=b)
> SPLProd.nest <- list(Areas = "Factories", Parts = c("Factories", "Areas"))
> SPLProd.lay <- fac.layout(unrandomized = SPLProd.unit,
+                           nested.factors = SPLProd.nest,
+                           randomized = SPLProd.ran, seed = 1025)
> SPLProd.lay
```

	Units	Permutation	Factories	Areas	Parts	Methods	Sources
1	1	32	1	I	1	2	C
2	2	33	1	I	2	2	A
3	3	31	1	I	3	2	B
4	4	35	1	II	1	3	B
5	5	34	1	II	2	3	A
6	6	36	1	II	3	3	C
7	7	30	1	III	1	1	B
8	8	28	1	III	2	1	C
9	9	29	1	III	3	1	A
10	10	9	2	I	1	2	C
11	11	7	2	I	2	2	B
12	12	8	2	I	3	2	A
13	13	2	2	II	1	1	A
14	14	3	2	II	2	1	B
15	15	1	2	II	3	1	C
16	16	5	2	III	1	3	A
17	17	4	2	III	2	3	B
18	18	6	2	III	3	3	C
19	19	26	3	I	1	2	B
20	20	25	3	I	2	2	A
21	21	27	3	I	3	2	C
22	22	20	3	II	1	3	C
23	23	19	3	II	2	3	B
24	24	21	3	II	3	3	A
25	25	24	3	III	1	1	B
26	26	23	3	III	2	1	A
27	27	22	3	III	3	1	C
28	28	13	4	I	1	3	B
29	29	14	4	I	2	3	C
30	30	15	4	I	3	3	A
31	31	12	4	II	1	1	C
32	32	11	4	II	2	1	A
33	33	10	4	II	3	1	B
34	34	16	4	III	1	2	B
35	35	17	4	III	2	2	A
36	36	18	4	III	3	2	C

9. Incomplete block designs

The expressions for obtaining a randomized layout for an incomplete block design are a modification of the expressions for randomizing an RCBD to take into account that there are now *k*, instead of *t*, units per block. Also, you will have to set up a factor, say Treat, that contains the design obtained from a textbook. The general R expressions are given below. To use these expressions to generate a layout for a particular case, you will need to assign values to *b*, *k* and *t* prior to using them and to substitute the actual names for *Blocks*, *Units, Treats* and *BIBD*. Also, the labels for the treatments are optional.

```
BIBD.unit <- list(Blocks=b, Plots=k)
BIBD.nest <- list(Plots = "Blocks")
Treats <- factor(c(the unrandomized layout for the BIBD),
                 labels=c("A","B" …))
BIBD.lay <- fac.layout(unrandomized = BIBD.unit,
                       nested.factors = BIBD.nest,
                       randomized = Treats, seed = 987)
remove("Treats")
BIBD.lay
```

Example: BIBD for four treatments in blocks of three

The following expressions obtain the randomized layout for a BIBD with $b = 4$, $k = 3$ and $t = 4$.

```
b <- 4
k <- 3
t <- 4
BIBD.unit <- list(Blocks=b, Plots=k)
BIBD.nest <- list(Plots = "Blocks")
BIBD.ran <- Treat <- factor(c(1,2,3, 1,2,4, 1,3,4, 2,3,4),
                                                  labels=c("A","B","C","D"))
BIBD.lay <- fac.layout(unrandomized = BIBD.unit, nested.factors = BIBD.nest,
                     randomized = Treat, seed = 987)
remove("Treats")
BIBD.lay
```

10. Balanced lattice square designs

The expressions for obtaining a randomized layout for a balanced lattice square design are a modification of the expressions for randomizing a set of Latin squares (case 3) to take into account that there are now *k*, instead of *t*, rows and columns. Also, you will have to set up a factor, say Treat, that contains the design obtained from a textbook. The general expressions are given below. To use these expressions to generate a layout for a particular case, you will need to assign values to *k* prior to using them and to substitute the actual names for *Squares*, *Rows, Columns*, *Treats* and the design object to contain them. Also, the labels for the treatments are optional.

```
#
# Balanced lattice squares
#
r <- (k+1)/2
t <- k*k
BalLattSq.unit <- list(Squares=r, Rows=k, Cols=k)
BalLattSq.nest <- list(Rows = "Squares", Cols = "Squares")
Treats <- factor(c(1,2,3, 4,5,6, 7,8,9, 1,6,8, 9,2,4, 5,7,3))
Treats <- factor(c(the unrandomized layout for the BLS),
                 labels=c("A","B" …))
BalLattSq.lay <- fac.layout(unrandomized = BalLattSq.unit,
                            nested.factors = BalLattSq.nest,
                            randomized = Treats, seed = 419)
remove("Treats")
BalLattSq.lay
```

Example: 3 □3 balanced lattice square

The following expressions obtain the randomized layout for a 3 ´ 3 balanced lattice square ($k = 3$).

```
#
# Balanced lattice squares
#
k <- 3
r <- (k+1)/2
t <- k*k
BalLattSq.unit <- list(Squares=r, Rows=k, Cols=k)
BalLattSq.nest <- list(Rows = "Squares", Cols = "Squares")
Treats <- factor(c(1,2,3, 4,5,6, 7,8,9, 1,6,8, 9,2,4, 5,7,3))
BalLattSq.lay <- fac.layout(unrandomized = BalLattSq.unit,
                            nested.factors = BalLattSq.nest,
                            randomized = Treats, seed = 419)
remove("Treats")
BalLattSq.lay
```

11. Youden square designs

The expressions for obtaining a randomized layout for a Youden square design are a modification of the expressions for randomizing a Latin square to take into account that there are now *k*, instead of *t*, columns. Also, you will have to set up a factor, say Treat, that contains the design obtained from a textbook. The general expressions are given below. To use these expressions to generate a layout for a particular case, you will need to assign values to *n*, *k* and *t* prior to using them and to substitute the actual names for *Rows, Columns*, *Treats* and *YDN*. Also, the labels for the treatments are optional.

```
YDN.unit <- list(Rows=t, Columns=k)
Treats <- factor(c(the unrandomized layout for the YS),
                 labels=c("A","B" …))
YDN.lay <- fac.layout(unrandomized = YDN.unit,
                      randomized = Treats, seed = 859)
remove("Treats")
YDN.lay
```

Example: Wear testing

The following expressions obtain the randomized layout for a 7 ´ 4 Youden square (Box, Hunter and Hunter, 2005, section 4.5):

```
t <- 7
k <- 4
n <- k*t
YDNWear.unit <- list(Machines=t, Positions=k)
Fabrics <- factor(c(3,5,6,7, 4,6,7,1, 5,7,1,2, 6,1,2,3, 7,2,3,4, 1,3,4,5,
                    2,4,5,6), labels=c("A","B","C","D","E","F","G"))
YDNWear.lay <- fac.layout(unrandomized = YDNWear.unit,
                          randomized = Fabrics, seed = 859)
remove("Fabrics")
YDNWear.lay
```

12. Power and sample size for designed experiments

In power and sample size calculations, in addition to specifying delta, sigma, power and alpha, one has to supply a number of quantities that vary with the design of the experiment. The following table summarizes these for the common designs, giving the degrees of freedom of the denominator as a function of *r*. Note that rm is the number of replicates in means being compared. For treatment means, this will be the product of the pure replication of the treatments, *r*, times a multiple, *m*, for the product of the number of levels of factors not involved in means being compared.

Design	*m*	*rm*	*df.num* (ν_1)	*df.denom* (ν_2)
CRD	1	r	$t-1$	$t(r-1)$
RCBD	1	b	$t-1$	$(t-1)(b-1)$
LS	1	$r(=t)$	$r-1$	$(r-1)(r-2)$
Factorial				
A	b	br	$a-1$	CRD $ab(r-1)$,
B	a	ar	$b-1$	RCBD $(ab-1)(r-1)$
A:B	1	r	$(a-1)(b-1)$	or LS $(r-1)(r-2)$
Standard split-plot				
A	b	br	$a-1$	$(a-1)(r-1)$
B	a	ar	$b-1$	$a(b-1)(r-1)$
A:B	1	r	$(a-1)(b-1)$	$a(b-1)(r-1)^{\dagger}$

[†]only approximate for effects not at the same level of A

a) Computing the power for given sample size

The function `power.exp` from the *dae* library is used for computing the power in detecting the difference between means for some, not necessarily proper, subset of the factors from a designed experiment.

The usage and arguments for this function are:

```
power.exp(rm=5,df.num=1, df.denom=10, delta=1, sigma=1,
   alpha=0.05, print=FALSE)
```

`rm`: the number of observations used in computing a mean.
`df.num`: the degrees of freedom of the numerator of the F for testing the term involving the means;
`df.denom`: the degrees of freedom of the denominator of the F for testing the term involving the means;
`delta`: the true difference between a pair of means;
`sigma`: population standard deviation;
`alpha`: the significance level to be used
`print`: T or F to have or not have a table of power calculation details printed out.

Note that the values given for the arguments in the above expression for `power.exp` are the default values assigned to the arguments if they are not set in a call to the function.

Example: Penicillin yield

Suppose it was expected that the minimum difference between a pair of treatment means is 5 and that $\alpha = 0.05$. In the analysis of variance for this experiment, the Residual MSq was 18.83 so we will take $\sigma^2 \approx 20$. Also $r = 5$ and $m = 1$. The output from the `power.exp` call to compute the power is given below. Note that alpha is not set in this call and so the default value of 0.05 will be used. Also, the expressions `3 * (rm - 1)` and `sqrt(20)` will be evaluated prior to the call to the function. To get the correct value of `rm` used in evaluating the expression `3 * (rm - 1`, `rm` needs to be set prior to calling `power.exp`.

```
> rm <- 5
> power.exp(rm=rm, df.num=3, df.denom=3*(rm-1), delta=5, sigma=sqrt(20),
+                                                          print=TRUE)
  rm df.num df.denom alpha delta    sigma lambda      powr
1  5      3       12  0.05     5 4.472136  3.125 0.2159032
[1] 0.2159032
```

b) Computing the sample size to achieve specified power

The function `no.reps` from the *dae* library is used to compute the required number of pure replicates, *r*, of the treatments in a designed experiment to achieve a specified power in detecting a difference between the means for some, not necessarily proper, subset of the treatment factors. The usage and arguments for this function are as follows:

```
no.reps(multiple=1, df.num=1,
                  df.denom=expression((df.num+1)*(r-1)),
                  delta=1, sigma=1, alpha=0.05, power =0.8,
                  tol = 0.025, print=FALSE)
```

`multiple`: the multiplier, *m*, which when multiplied by the number of pure replicates of a treatment, *r*, gives the number of observations (rm) used in computing means for the treatment factor subset; *m* is the replication arising from other treatment factors. However, for single treatment factor experiments the subset can only be the treatment factor and $m = 1$

`df.num`: the degrees of freedom of the numerator of the F for testing the term involving the treatment factor subset;

`df.denom`: an expression for the degrees of freedom of the denominator of the F for testing the term involving the treatment factor subset — it must involve *r*, the number of pure replicates, can involve other arguments to `no.reps` such as `multiple` and `df.num`, and must be enclosed in an `expression` function so that it is not evaluated when `no.reps` is called but will be evaluated as different values of *r* are tried during execution of `no.reps`;

`delta`: the true difference between a pair of means for some, not necessarily proper, subset of the treatment factors;

`sigma`: population standard deviation;

`alpha`: the significance level to be used;

`power`: the minimum power to be achieved;

`tol`: the maximum difference tolerated between the `power` required and the power computed in determining the number of replicates;

`print`: T or F to have or not have a table of power calculation details printed out.

Example II.1.Penicillin yield (continued)

We now determine the number of replicates required to achieve a power of 0.80 in detecting $\Delta = 5$ with $\alpha = 0.05$. We continue to take $\sigma^2 \approx 20$. The output from the use of `no.reps` is as follows:

```
> no.reps(multiple=1, df.num=3, df.denom=expression(df.num*(r-1)), delta=5,
+ sigma=sqrt(20), power=0.8, print=FALSE)
$nreps
[1] 19

$power
[1] 0.8055926
```

13. References

Box, G. E. P., Hunter, J.S. and Hunter, W.G. (2005). *Statistics for experimenters: design, innovation, and discovery*. Hoboken, N.J., Wiley-Interscience.

Johnson, N. L. and F. C. Leone (1964). *Statistics and experimental design in engineering and the physical sciences*. New York, Wiley.