Package 'randomizr'

August 27, 2026

Title Easy-to-Use Tools for Common Forms of Random Assignment and Sampling

Version 2.0.1

Description Generates random assignments for common experimental designs and random samples for common sampling designs.

URL https://declaredesign.org/r/randomizr/, https://github.com/DeclareDesign/randomizr

BugReports https://github.com/DeclareDesign/randomizr/issues

Imports Rcpp, stats, utils

LinkingTo Rcpp

Depends R (>= 3.6.0)

License MIT + file LICENSE

Encoding UTF-8

Suggests knitr, dplyr, tidyr, purrr, readr, ggplot2, blockTools, estimatr, DeclareDesign, fabricatr, BalancedSampling, sampling, testthat, rmarkdown

VignetteBuilder knitr

Config/roxygen2/version 8.0.0

NeedsCompilation yes

Author Alexander Coppock [aut, cre] (ORCID: <https://orcid.org/0000-0002-5733-2386>), Jasper Cooper [ctb] (ORCID: <https://orcid.org/0000-0002-8639-3188>), Neal Fultz [ctb] (C version of restricted partitions), Graeme Blair [ctb] (ORCID: <https://orcid.org/0000-0001-9164-2102>), Macartan Humphreys [ctb] (ORCID: <https://orcid.org/0000-0001-7029-2326>)

Maintainer Alexander Coppock <acoppock@gmail.com>

Repository CRAN

Date/Publication 2026-08-27 05:10:15 UTC

Contents

`randomizr-package` *randomizr: Easy-to-Use Tools for Common Forms of Random Assignment and Sampling*

Description

randomizr generates random assignments for common experimental designs and random samples for common sampling designs. The functions are named for the procedure they implement, and each has a '_probabilities' companion that returns the probability of each unit falling into each condition, which is what inverse-probability weights are built from.

Random assignment

- [simple_ra()] assigns each unit independently, so the number treated varies from draw to draw.
- [complete_ra()] fixes the number treated on every draw.
- [block_ra()] conducts complete assignment separately within blocks of similar units, which increases precision.
- [cluster_ra()] assigns whole groups together, for interventions that cannot be delivered to individuals.
- [block_and_cluster_ra()] does both at once.
- [balanced_ra()] (experimental) holds condition counts (and, with `formula`, covariate totals) at their targets while keeping each unit's probability exact.
- [declare_ra()] describes a design once so it can be reused by [conduct_ra()] to draw assignments and by [obtain_condition_probabilities()] to recover the probabilities. Balanced assignment is opt-in: `ra_type = "balanced"`, `prob_unit_each`, or `formula`.

Random sampling

The sampling functions mirror the assignment ones: [simple_rs()], [complete_rs()], [strata_rs()], [cluster_rs()] and [strata_and_cluster_rs()], with [declare_rs()], [draw_rs()] and [obtain_inclusion_probabilities()] playing the roles that [declare_ra()], [conduct_ra()] and [obtain_condition_probabilities()] play for assignment.

Randomization inference

[obtain_permutation_matrix()] enumerates or samples the assignments a design could have produced, and [obtain_num_permutations()] counts them.

Author(s)

Maintainer: Alexander Coppock <acoppock@gmail.com> (ORCID)

Authors:

- Alexander Coppock <acoppock@gmail.com> (ORCID)

Other contributors:

- Jasper Cooper <jaspercooper@gmail.com> (ORCID) [contributor]
- Neal Fultz <nfultz@gmail.com> (C version of restricted partitions) [contributor]
- Graeme Blair <graeme.blair@gmail.com> (ORCID) [contributor]
- Macartan Humphreys <macartan@gmail.com> (ORCID) [contributor]

References

Blair, G., Cooper, J., Coppock, A. and Humphreys, M. (2019). Declaring and Diagnosing Research Designs. *American Political Science Review* 113(3), 838-859. doi:10.1017/S0003055419000194

Gerber, A. S. and Green, D. P. (2012). *Field Experiments: Design, Analysis, and Interpretation*. New York: W. W. Norton.

See Also

Useful links:

- https://declaredesign.org/r/randomizr/
- https://github.com/DeclareDesign/randomizr
- Report bugs at https://github.com/DeclareDesign/randomizr/issues

Examples

```
# Complete random assignment: exactly 50 of 100 units treated, every draw.
Z <- complete_ra(N = 100, m = 50)
table(Z)

# Blocking on a covariate usually buys precision.
blocks <- rep(c("small", "large"), times = c(60, 40))
Z <- block_ra(blocks = blocks)
table(blocks, Z)

# Declare once, then draw and recover probabilities from the same object.
declaration <- declare_ra(N = 100, m = 50)
Z <- conduct_ra(declaration)
probs <- obtain_condition_probabilities(declaration, Z)
table(probs)
```

`balanced_ra` *Random assignment with tight targets*

Description

Experimental. `balanced_ra` draws random assignment with tight targets: condition counts at the floor or ceiling of what the probabilities imply, and, with `formula`, covariate totals too. Each unit's probability stays exact. That is useful when probabilities vary across units, and also when they do not: leftover pairing keeps two-arm blocked counts tight overall as well as within each block, and cube-on-X balances a continuous covariate without binning it.

Usage

```
balanced_ra(
  N = NULL,
  prob = NULL,
  prob_unit = NULL,
  prob_unit_each = NULL,
  blocks = NULL,
  clusters = NULL,
  num_arms = NULL,
  conditions = NULL,
  formula = NULL,
```

```
  check_inputs = TRUE,
  .X = NULL
)
```

Arguments

N	The number of units. Optional when formula or the length of prob_unit (or blocks or clusters) identifies N. A single positive integer. If supplied it must match. (optional)
prob	A single number between 0 and 1: the probability of assignment to treatment, shared by every unit, for a two-arm design. Defaults to 0.5 when no probability argument is supplied, so balanced_ra(4) is complete assignment of four units. Supply exactly one of prob, prob_unit and prob_unit_each. (optional)
prob_unit	A numeric vector of length N giving each unit's probability of assignment to treatment, for a two-arm design. Unlike elsewhere in randomizr these need not be equal across units. A single number is refused, since that is what prob is for. Supply exactly one of prob, prob_unit and prob_unit_each. (optional)
prob_unit_each	A numeric matrix with one row per unit and one column per condition, giving each unit's probability of assignment to each condition, for a multi-arm design. Rows must sum to 1. Supply exactly one of prob, prob_unit and prob_unit_each. (optional)
blocks	A vector of length N indicating which block each unit belongs to. When supplied, two-arm counts are held tight within each block and overall; with three or more arms the tight counts are the within-block ones. (optional)
clusters	A vector of length N indicating which cluster each unit belongs to. Whole clusters are assigned together, so the probabilities must be the same for every unit in a cluster, and the tight counts become counts of clusters rather than of units. May be combined with blocks, in which case every cluster must sit entirely inside one block. May also be combined with formula, in which case each cluster's covariates are the averages of its units' covariates, so that a cluster counts once however many units it holds and the treated count that is held tight remains a count of clusters. (optional)
num_arms	The number of treatment arms. Inferred when omitted. Supplied without any probability argument, num_arms (or conditions) of three or more expands to equal-probability assignment, as in complete_ra(). (optional)
conditions	A vector giving the names of the conditions. (optional)
formula	A model formula whose model matrix is the balancing matrix X in the cube method, e.g. ~ x + B. The intercept column is the count constraint; ~ 0 + x drops it and the treated count may wander. Names are looked up where the formula was written, then in the calling frame, so the usual dat |> mutate(Z = balanced_ra(formula = ~ x)) finds the column x. Two-arm only. May be combined with clusters; cannot be combined with blocks or prob_unit_each. (optional)
check_inputs	Logical. Whether to verify before assigning that the arguments are internally consistent: that probabilities lie between 0 and 1, that rows of a probability matrix sum to 1, that probabilities are constant within a cluster, and that clusters

nest within blocks. Defaults to `TRUE`. Set to `FALSE` to skip the checks when drawing many assignments from probabilities that have already been verified. (optional)

`.X` Internal. A balancing matrix already built from `formula`, supplied by `declare_ra()` so that the formula's variables are looked up once, when the design is declared, rather than on every draw. Not for direct use. (optional)

Details

With unit-varying probabilities it fills the gap between `simple_ra()`, which honors those probabilities but lets the number treated wander, and `complete_ra()`, which fixes the number treated but requires every unit to share the same probability.

The "balanced" in the name is balanced sampling in the sense of Deville and Tillé (2004). With the default arguments the realized counts are held against their targets. Pass `formula` to add linear balancing constraints on covariates (cube-on-X): the flight keeps $X'Z$ near $X'\pi$. Landing may drop a constraint, so exact tightness on every column is not always possible. `blocks` is a different device: it tightens counts inside discrete groups. The two cannot be combined.

Two motivating cases: a race in which contestants have unequal chances and exactly one must win; and two districts of three villages, three to treat, blocked by district, so that each district should receive one or two and the total should be three.

Value

A vector of length N giving the condition of each unit. As in `complete_ra()`: integer 0/1 in a two-arm design, unless `num_arms` or `conditions` is supplied explicitly, in which case a factor ordered by `conditions`; a factor in a multi-arm design.

What is guaranteed

Every unit receives exactly one condition. Each unit's probability of each condition is the probability supplied. Counts are tight within each block always, and tight overall as well when there are two arms. With three or more arms and `blocks`, the overall count can wander; see the vignette *Introduction to balanced_ra*. With `clusters`, the tight counts are counts of clusters. With `formula`, first-order inclusion probabilities remain exact; covariate totals are as close as the landing phase allows. See that vignette.

Tight counts have one exception, and it is an arithmetic one rather than a design one. Each step of the algorithm is sized so that at least one unit lands exactly on 0 or on 1. Every so often rounding error in floating-point arithmetic leaves every unit in that step a hair short of its bound, and the function then settles the unit with the least room left by a coin weighted by the value that unit currently holds. That coin keeps the unit's assignment probability exactly right, so the probability guarantee is untouched. It does not respect the count, so a draw that reaches this fallback can finish one unit away from the floor or the ceiling. We have not been able to make it happen: it did not arise in any of several thousand draws across dozens of randomly generated designs. It is documented because it is reachable in principle, not because it is expected in practice.

Balance when probabilities vary

The cube holds $X'Z$ near $X'\pi$, which is the treated total of each balancing column against the total its assignment probabilities imply. When every unit shares a probability, that target amounts

to splitting the column evenly between the arms, and `formula` does what its name suggests. When probabilities vary from unit to unit, the two targets come apart.

Suppose p_i rises with x_i. High-x units are meant to be treated more often, so the treated group ought to have the higher mean of x, and it does: the average treated-minus-control difference in x under `formula = ~ x` is the same one `simple_ra()` gives on the same probabilities. What the cube tightens is the spread of that difference around its target, and with it the Horvitz-Thompson residual for the x total.

In short, `formula` does not equalize the arms when p_i varies, and it is not meant to. Weight by the reciprocal of the assignment probability, as for any unequal-probability design; `balanced_ra_probabilities()` returns the probabilities to weight by. With a constant p the question does not arise.

Order of the covariates

The flight phase sorts units by the first column of X that is not constant and works through them in a sliding window, so each step pairs units with nearby values of that column. An intercept is a column of ones and so is passed over, which makes the sort column `x` under `~ x` and `x1` under `~ 0 + x1 + x2`. The design therefore balances smooth functions of that first covariate and not only its linear total: in simulations at $N = 200$ with a constant p, the treated-minus-control spread in x^2 and x^3 runs several times tighter than under `complete_ra()`, though how much tighter varies with the covariate draw, and a heavy-tailed x narrows the gain.

The gain is also uneven. Only one column drives the sort, so under `~ x1 + x2` the spread in x_1^2 tightens while the spread in x_2^2 stays about where complete assignment leaves it. Both linear totals are held tight. A covariate you name but do not put first is balanced in its own right and in nothing else, and a covariate you do not name at all is not balanced.

Sorting is a choice made here rather than a feature of the cube method, which constrains only the linear span of X. Put the covariate whose relationship with the outcome you least trust yourself to model first in the formula.

Analyzing the result

When p_i varies across units, an unweighted comparison of means is not the average treatment effect. Weight each unit by the reciprocal of the probability of the condition it landed in; `balanced_ra_probabilities()` returns the matrix of probabilities those weights are built from, in the same form as the other `_probabilities` functions in randomizr.

Standard errors then divide into two cases, and the vignette *Introduction to balanced_ra* measures both.

On the count-tight designs, meaning every call that does not pass `formula`, the usual heteroskedasticity-consistent intervals behave about as they do after `complete_ra()`. Holding counts tight makes assignments negatively dependent across units, which is a reason to ask the question, but in simulation it did not move HC2 coverage appreciably away from its nominal rate for two-arm, blocked two-arm or three-arm designs.

With `formula` it is different. The design removes assignment variance that the variance estimator cannot see, so the reported interval is wider than the estimator's true sampling variability warrants. At $N = 200$ with a strongly prognostic x, HC2 on an unadjusted regression covered the true effect on every draw, with an average standard error well over twice the estimator's actual standard deviation. That is valid but wasteful: it discards the precision the design was chosen to buy. Fitting

Lin's estimator on the same columns recovers most of it, and stops recovering it when the adjustment model is wrong, so the case for this design is strongest exactly where the reported interval understates the gain. Adjusting linearly for x when the outcome was quadratic in it, for instance, returned coverage to 1.000 with the standard error again more than twice too large.

`estimatr::horvitz_thompson()` is conservative here for a related reason, and an exact variance is not a missing feature so much as an open problem: the joint inclusion probabilities of a cube design have no closed form. That is what Deville and Tillé (2005) approximate, and randomizr does not implement that approximation.

Experimental

This function is new in randomizr 2.0.1 and its interface may change. Declare a design with `declare_ra()` by setting `ra_type = "balanced"` or by supplying `prob_unit_each` or `formula`; `conduct_ra()` and `obtain_condition_probabilities()` then dispatch here. The vignette *Introduction to balanced_ra* has the count-tight algorithm and a four-unit cube-on-X walk-through.

References

Deville, J.-C. and Tillé, Y. (2004). Efficient balanced sampling: the cube method. *Biometrika* 91(4), 893-912. doi:10.1093/biomet/91.4.893

Deville, J.-C. and Tillé, Y. (1998). Unequal probability sampling without replacement through a splitting method. *Biometrika* 85(1), 89-101. doi:10.1093/biomet/85.1.89

Chauvet, G. and Tillé, Y. (2006). A fast algorithm for balanced sampling. *Computational Statistics* 21(1), 53-62. doi:10.1007/s0018000602502

Deville, J.-C. and Tillé, Y. (2005). Variance approximation under balanced sampling. *Journal of Statistical Planning and Inference* 128(2), 569-591. doi:10.1016/j.jspi.2003.11.011

See Also

`balanced_ra_probabilities()`, `complete_ra()`, `block_ra()`, `simple_ra()`, the vignette *Introduction to balanced_ra*

Examples

```
# Four units, default probability 0.5: complete assignment of two treated.
table(balanced_ra(4))

# A race between contestants with unequal chances, in which exactly one wins
# because the chances sum to 1.
chances <- c(0.5, 0.3, 0.15, 0.05)
winners <- replicate(1000, which(balanced_ra(prob_unit = chances) == 1))
table(winners) / 1000     # close to chances

# Unequal probabilities, two arms, with the number treated held tight.
p <- c(0.2, 0.4, 0.6, 0.8, 0.5, 0.5)
Z <- balanced_ra(prob_unit = p)
table(Z)

# Repeating the draw: probabilities are honored, and exactly 3 are treated
```

```
# every time because the probabilities sum to 3.
reps <- replicate(1000, balanced_ra(prob_unit = p))
rowMeans(reps)          # close to p
table(colSums(reps))    # always 3

# Two districts of three villages, three to be treated, blocked by district.
# Each district gets one or two; the total is always three.
districts <- rep(c("north", "south"), each = 3)
reps <- replicate(1000, balanced_ra(blocks = districts))
table(colSums(reps))                          # always 3
table(colSums(reps[districts == "north", ]))   # 1 or 2

# Three arms with unit-varying probabilities.
P <- cbind(c(0.15, 0.47), c(0.65, 0.48), c(0.20, 0.05))
table(replicate(1000, balanced_ra(prob_unit_each = P))[1, ])

# Whole clusters assigned together, with unequal cluster probabilities. The
# number of treated clusters is fixed; the number of treated units is not,
# because the clusters differ in size.
clusters <- rep(1:6, times = c(3, 1, 4, 2, 5, 3))
p_cluster <- c(0.2, 0.4, 0.6, 0.8, 0.5, 0.5)
Z <- balanced_ra(prob_unit = p_cluster[clusters], clusters = clusters)
table(clusters, Z)

# Blocks and clusters together: a tight number of treated clusters in each
# block.
blocks <- ifelse(clusters <= 3, "east", "west")
Z <- balanced_ra(prob = 0.5, clusters = clusters, blocks = blocks)
table(blocks, Z)

# Cube-on-X: keep the treated total of a continuous covariate near its
# target. The intercept in ~ x is the count constraint. N is inferred
# from the looked-up formula variables.
x <- c(1, 2, 3, 6)
Z <- balanced_ra(formula = ~ x)
sum(x * Z)   # near 6

# Cube-on-X with clusters. Each cluster is treated as one unit carrying the
# average of its members' covariates, so three of the six clusters are
# treated on every draw and it is the cluster means of x that are balanced.
x_cl <- c(-2, -1, 0, 1, 2, 3)[clusters]
Z <- balanced_ra(prob = 0.5, clusters = clusters, formula = ~ x_cl)
table(clusters, Z)
```

---

balanced_ra_probabilities

Probabilities of assignment: Balanced Random Assignment

Description

Experimental. Returns the probability that each unit is assigned to each condition under `balanced_ra()`. Because those probabilities are supplied by the caller rather than derived from a design, this function mainly validates and normalizes them into the matrix form the other `_probabilities` functions return.

Usage

```
balanced_ra_probabilities(
  N = NULL,
  prob = NULL,
  prob_unit = NULL,
  prob_unit_each = NULL,
  blocks = NULL,
  clusters = NULL,
  num_arms = NULL,
  conditions = NULL,
  formula = NULL,
  check_inputs = TRUE
)
```

Arguments

`N`	The number of units. Optional when `formula` or the length of `prob_unit` (or `blocks` or `clusters`) identifies N. A single positive integer. If supplied it must match. (optional)
`prob`	A single number between 0 and 1: the probability of assignment to treatment, shared by every unit, for a two-arm design. Defaults to 0.5 when no probability argument is supplied, so `balanced_ra(4)` is complete assignment of four units. Supply exactly one of `prob`, `prob_unit` and `prob_unit_each`. (optional)
`prob_unit`	A numeric vector of length N giving each unit's probability of assignment to treatment, for a two-arm design. Unlike elsewhere in randomizr these need not be equal across units. A single number is refused, since that is what `prob` is for. Supply exactly one of `prob`, `prob_unit` and `prob_unit_each`. (optional)
`prob_unit_each`	A numeric matrix with one row per unit and one column per condition, giving each unit's probability of assignment to each condition, for a multi-arm design. Rows must sum to 1. Supply exactly one of `prob`, `prob_unit` and `prob_unit_each`. (optional)
`blocks`	A vector of length N indicating which block each unit belongs to. When supplied, two-arm counts are held tight within each block and overall; with three or more arms the tight counts are the within-block ones. (optional)
`clusters`	A vector of length N indicating which cluster each unit belongs to. Whole clusters are assigned together, so the probabilities must be the same for every unit in a cluster, and the tight counts become counts of clusters rather than of units. May be combined with `blocks`, in which case every cluster must sit entirely inside one block. May also be combined with `formula`, in which case each cluster's covariates are the averages of its units' covariates, so that a cluster counts once

however many units it holds and the treated count that is held tight remains a count of clusters. (optional)

`num_arms` The number of treatment arms. Inferred when omitted. Supplied without any probability argument, `num_arms` (or `conditions`) of three or more expands to equal-probability assignment, as in `complete_ra()`. (optional)

`conditions` A vector giving the names of the conditions. (optional)

`formula` A model formula whose model matrix is the balancing matrix X in the cube method, e.g. `~ x + B`. The intercept column is the count constraint; `~ 0 + x` drops it and the treated count may wander. Names are looked up where the formula was written, then in the calling frame, so the usual `dat |> mutate(Z = balanced_ra(formula = ~ x))` finds the column `x`. Two-arm only. May be combined with `clusters`; cannot be combined with `blocks` or `prob_unit_each`. (optional)

`check_inputs` Logical. Whether to verify before assigning that the arguments are internally consistent: that probabilities lie between 0 and 1, that rows of a probability matrix sum to 1, that probabilities are constant within a cluster, and that clusters nest within blocks. Defaults to `TRUE`. Set to `FALSE` to skip the checks when drawing many assignments from probabilities that have already been verified. (optional)

Details

These are the quantities inverse-probability weights are built from: weight each unit by the reciprocal of the probability of the condition it landed in.

Value

A matrix of probabilities of assignment, one row per unit and one column per condition, with columns named `prob_<condition>`.

See Also

`balanced_ra()`

Examples

```
balanced_ra_probabilities(prob_unit = c(0.2, 0.4, 0.6, 0.8, 0.5, 0.5))
```

---

block_and_cluster_ra *Blocked and Clustered Random Assignment*

Description

`block_and_cluster_ra` assigns whole clusters to conditions, conducting the assignment separately within each block. Use it when treatment can only be delivered to a group and the groups differ in ways worth balancing on. Clustering costs precision, since the effective sample size is the number of clusters rather than the number of units; blocking buys some of it back by guaranteeing treated and control clusters within every block.

Usage

```
block_and_cluster_ra(
  blocks = NULL,
  clusters = NULL,
  prob = NULL,
  prob_unit = NULL,
  prob_each = NULL,
  m = NULL,
  m_unit = NULL,
  block_m = NULL,
  block_m_each = NULL,
  block_prob = NULL,
  block_prob_each = NULL,
  num_arms = NULL,
  conditions = NULL,
  check_inputs = TRUE
)
```

Arguments

blocks	A vector of length N indicating which block each unit belongs to. Every unit in a cluster must belong to the same block. (required)
clusters	A vector of length N indicating which cluster each unit belongs to. (required)
prob	Use for a two-arm design in which either `floor(N_clusters_block*prob)` or `ceiling(N_clusters_block*prob)` clusters are assigned to treatment within each block. Which of the two is used is itself random: the ceiling is drawn with probability equal to the fractional part of `N_clusters_block*prob` and the floor otherwise, which makes each cluster's probability of assignment exactly `prob`. Must be a real number between 0 and 1 inclusive. (optional)
prob_unit	Use for a two-arm design. Must be of length N. `tapply(prob_unit, blocks, unique)` will be passed to `block_prob`, so it must be constant within each block. (optional)
prob_each	Use for a multi-arm design. A numeric vector giving the probability of assignment to each condition. All entries must be between 0 and 1 inclusive and must sum to 1. Because of integer rounding, the exact number of clusters assigned to each condition may differ slightly from assignment to assignment, but the overall probability of assignment is exactly `prob_each`. (optional)
m	Use for a two-arm design in which the scalar `m` gives the fixed number of clusters assigned to treatment within every block. This count does not vary across blocks. (optional)
m_unit	Use for a two-arm design. Must be of length N. `tapply(m_unit, blocks, unique)` will be passed to `block_m`, so it must be constant within each block. (optional)
block_m	Use for a two-arm design in which `block_m` gives the number of clusters to assign to treatment within each block. Must be a numeric vector as long as the number of blocks, in the same order as `sort(unique(blocks))`. (optional)

`block_m_each` Use for a multi-arm design in which `block_m_each` gives the number of clusters assigned to each condition within each block. Must be a matrix with one row per block and one column per treatment arm. Rows respect the ordering of blocks by `sort(unique(blocks))`; columns should be in the order of `conditions`, if specified. (optional)

`block_prob` Use for a two-arm design in which the probability of assignment to treatment varies across blocks. Must be in the same order as `sort(unique(blocks))`. Differs from `prob` in that the probability of assignment can vary across blocks. (optional)

`block_prob_each`
Use for a multi-arm design in which assignment probabilities vary across blocks. Must be a matrix with one row per block and one column per treatment arm; each row must sum to 1. Rows respect the ordering of `sort(unique(blocks))`. Use only if the probabilities of assignment should vary by block, otherwise use `prob_each`. (optional)

`num_arms` The number of treatment arms. If unspecified, determined from the other arguments. (optional)

`conditions` A character vector giving the names of the treatment groups. If unspecified, the treatment groups will be named 0 (for control) and 1 (for treatment) in a two-arm trial and T1, T2, T3, in a multi-arm trial. A two-group design in which `num_arms` is set to 2 will use condition names T1 and T2. (optional)

`check_inputs` Logical. Whether to verify before assigning that the arguments are internally consistent: that clusters nest within blocks, that counts sum to the number of clusters in each block, that probabilities lie between 0 and 1 and sum to 1, and so on. Defaults to `TRUE`. `FALSE` skips the checking only: `num_arms` and `conditions` are still derived from the other arguments, so the same call draws the same assignment either way. What goes is the verification, and an impossible design is then no longer refused. `block_m` larger than a block, for instance, quietly treats the whole block. Declaring the design once with `declare_ra()` and drawing from it with `conduct_ra()` is the usual way to avoid re-checking the same arguments in a simulation. (optional)

Details

Clusters must nest within blocks: every unit in a cluster has to belong to the same block.

Value

A vector of length N indicating the treatment condition of each unit. Every unit in a cluster receives the same value. Numeric in a two-arm trial; a factor (ordered by `conditions`) in a multi-arm trial.

See Also

`cluster_ra()`, `block_ra()`, `strata_and_cluster_rs()`

Examples

```
# Twelve clusters, of sizes 1 through 12, nested in four blocks of three
```

```
clusters <- rep(letters[1:12], times = 1:12)

blocks <- rep(NA, length(clusters))
blocks[clusters %in% letters[1:3]] <- "block_1"
blocks[clusters %in% letters[4:6]] <- "block_2"
blocks[clusters %in% letters[7:9]] <- "block_3"
blocks[clusters %in% letters[10:12]] <- "block_4"


table(blocks, clusters)

Z <- block_and_cluster_ra(blocks = blocks,
                          clusters = clusters)

table(Z, blocks)
table(Z, clusters)

Z <- block_and_cluster_ra(blocks = blocks,
                          clusters = clusters,
                          num_arms = 3)

table(Z, blocks)
table(Z, clusters)

Z <- block_and_cluster_ra(blocks = blocks,
                          clusters = clusters,
                          prob_each = c(0.2, 0.5, 0.3))

# One row per block, one column per arm: how many clusters go where
block_m_each <- rbind(c(1, 2),
                      c(2, 1),
                      c(1, 2),
                      c(2, 1))

Z <- block_and_cluster_ra(blocks = blocks,
                          clusters = clusters,
                          block_m_each = block_m_each)

table(Z, blocks)
table(Z, clusters)
```

`block_and_cluster_ra_probabilities` *Probabilities of assignment: Blocked and Clustered Random Assignment*

Description

Returns the probability that each unit is assigned to each condition when clusters are assigned within blocks. Probabilities vary across blocks and are constant within a cluster.

Usage

```
block_and_cluster_ra_probabilities(
  blocks = NULL,
  clusters = NULL,
  prob = NULL,
  prob_unit = NULL,
  prob_each = NULL,
  m = NULL,
  m_unit = NULL,
  block_m = NULL,
  block_m_each = NULL,
  block_prob = NULL,
  block_prob_each = NULL,
  num_arms = NULL,
  conditions = NULL,
  check_inputs = TRUE
)
```

Arguments

`blocks`	A vector of length N indicating which block each unit belongs to. Every unit in a cluster must belong to the same block. (required)
`clusters`	A vector of length N indicating which cluster each unit belongs to. (required)
`prob`	Use for a two-arm design in which either `floor(N_clusters_block*prob)` or `ceiling(N_clusters_block*prob)` clusters are assigned to treatment within each block. Which of the two is used is itself random: the ceiling is drawn with probability equal to the fractional part of `N_clusters_block*prob` and the floor otherwise, which makes each cluster's probability of assignment exactly `prob`. Must be a real number between 0 and 1 inclusive. (optional)
`prob_unit`	Use for a two-arm design. Must be of length N. `tapply(prob_unit, blocks, unique)` will be passed to `block_prob`, so it must be constant within each block. (optional)
`prob_each`	Use for a multi-arm design. A numeric vector giving the probability of assignment to each condition. All entries must be between 0 and 1 inclusive and must sum to 1. Because of integer rounding, the exact number of clusters assigned to each condition may differ slightly from assignment to assignment, but the overall probability of assignment is exactly `prob_each`. (optional)
`m`	Use for a two-arm design in which the scalar `m` gives the fixed number of clusters assigned to treatment within every block. This count does not vary across blocks. (optional)
`m_unit`	Use for a two-arm design. Must be of length N. `tapply(m_unit, blocks, unique)` will be passed to `block_m`, so it must be constant within each block. (optional)
`block_m`	Use for a two-arm design in which `block_m` gives the number of clusters to assign to treatment within each block. Must be a numeric vector as long as the number of blocks, in the same order as `sort(unique(blocks))`. (optional)

`block_m_each` Use for a multi-arm design in which `block_m_each` gives the number of clusters assigned to each condition within each block. Must be a matrix with one row per block and one column per treatment arm. Rows respect the ordering of blocks by `sort(unique(blocks))`; columns should be in the order of `conditions`, if specified. (optional)

`block_prob` Use for a two-arm design in which the probability of assignment to treatment varies across blocks. Must be in the same order as `sort(unique(blocks))`. Differs from `prob` in that the probability of assignment can vary across blocks. (optional)

`block_prob_each`
Use for a multi-arm design in which assignment probabilities vary across blocks. Must be a matrix with one row per block and one column per treatment arm; each row must sum to 1. Rows respect the ordering of `sort(unique(blocks))`. Use only if the probabilities of assignment should vary by block, otherwise use `prob_each`. (optional)

`num_arms` The number of treatment arms. If unspecified, determined from the other arguments. (optional)

`conditions` A character vector giving the names of the treatment groups. If unspecified, the treatment groups will be named 0 (for control) and 1 (for treatment) in a two-arm trial and T1, T2, T3, in a multi-arm trial. A two-group design in which `num_arms` is set to 2 will use condition names T1 and T2. (optional)

`check_inputs` Logical. Whether to verify before assigning that the arguments are internally consistent: that clusters nest within blocks, that counts sum to the number of clusters in each block, that probabilities lie between 0 and 1 and sum to 1, and so on. Defaults to `TRUE`. `FALSE` skips the checking only: `num_arms` and `conditions` are still derived from the other arguments, so the same call draws the same assignment either way. What goes is the verification, and an impossible design is then no longer refused. `block_m` larger than a block, for instance, quietly treats the whole block. Declaring the design once with `declare_ra()` and drawing from it with `conduct_ra()` is the usual way to avoid re-checking the same arguments in a simulation. (optional)

Details

These are the quantities inverse-probability weights are built from: weight each unit by the reciprocal of the probability of the condition it landed in, which `obtain_condition_probabilities()` extracts for you.

Value

A matrix with N rows and one column per treatment condition, with columns named `prob_<condition>`. Entry (i, j) is the probability that unit i is assigned to condition j, and every row sums to 1.

See Also

`block_and_cluster_ra()`

Examples

```
clusters <- rep(letters[1:12], times = 1:12)
blocks <- rep(NA, length(clusters))
blocks[clusters %in% letters[1:3]] <- "block_1"
blocks[clusters %in% letters[4:6]] <- "block_2"
blocks[clusters %in% letters[7:9]] <- "block_3"
blocks[clusters %in% letters[10:12]] <- "block_4"


prob_mat <- block_and_cluster_ra_probabilities(clusters = clusters,
                                               blocks = blocks)
head(prob_mat)

prob_mat <- block_and_cluster_ra_probabilities(clusters = clusters,
                                               blocks = blocks,
                                               num_arms = 3)
head(prob_mat)

prob_mat <- block_and_cluster_ra_probabilities(clusters = clusters,
                                               blocks = blocks,
                                               prob_each = c(0.2, 0.5, 0.3))
head(prob_mat)

# One row per block, one column per arm: how many clusters go where
block_m_each <- rbind(c(1, 2),
                      c(2, 1),
                      c(1, 2),
                      c(2, 1))

prob_mat <- block_and_cluster_ra_probabilities(clusters = clusters,
                                               blocks = blocks,
                                               block_m_each = block_m_each)
head(prob_mat)
```

`block_ra` *Block Random Assignment*

Description

`block_ra` assigns units to treatment conditions within pre-defined groups called blocks (or strata). Within each block, complete random assignment determines which units are treated. Blocking typically reduces the sampling variability of an experiment relative to simple or complete random assignment: by guaranteeing that treated and control units are drawn from every covariate-defined subgroup, it rules out the unlucky assignments that would otherwise pull estimates far from the true average treatment effect. The precision gain is largest when the blocking variable is strongly correlated with potential outcomes; if the blocking variable is uncorrelated with outcomes, blocking neither helps nor hurts.

Usage

```
block_ra(
  blocks = NULL,
  prob = NULL,
  prob_unit = NULL,
  prob_each = NULL,
  m = NULL,
  m_unit = NULL,
  block_m = NULL,
  block_m_each = NULL,
  block_prob = NULL,
  block_prob_each = NULL,
  num_arms = NULL,
  conditions = NULL,
  check_inputs = TRUE,
  .block_int = NULL,
  .N_per_block = NULL
)
```

Arguments

blocks	A vector of length N indicating which block each unit belongs to. Can be character, factor, or numeric. (required)
prob	Use for a two-arm design in which either `floor(N_block*prob)` or `ceiling(N_block*prob)` units are assigned to treatment within each block. Which of the two is used is itself random: the ceiling is drawn with probability equal to the fractional part of `N_block*prob` and the floor otherwise, which makes each unit's probability of assignment exactly `prob`. When `N_block*prob` is a whole number the count is fixed. Must be a real number between 0 and 1. (optional)
prob_unit	Use for a two-arm design. Must be of length N. `tapply(prob_unit, blocks, unique)` will be passed to `block_prob`. (optional)
prob_each	Use for a multi-arm design in which the values of `prob_each` determine the probabilities of assignment to each treatment condition. Must be a numeric vector giving the probability of assignment to each condition. All entries must be nonnegative real numbers between 0 and 1 and the total must sum to 1. Because of integer rounding, the exact number of units assigned to each condition may differ slightly from assignment to assignment, but the overall probability of assignment is exactly `prob_each`. (optional)
m	Use for a two-arm design in which the scalar `m` gives the fixed number of units to assign to treatment within every block. This count does not vary across blocks. (optional)
m_unit	Use for a two-arm design. Must be of length N. `tapply(m_unit, blocks, unique)` will be passed to `block_m`. (optional)
block_m	Use for a two-arm design in which `block_m` gives the number of units to assign to treatment within each block. Must be a numeric vector as long as the number of blocks, in the same order as `sort(unique(blocks))`. (optional)

`block_m_each` Use for a multi-arm design in which `block_m_each` gives the number of units assigned to each condition within each block. Must be a matrix with one row per block and one column per treatment arm. Rows should respect the ordering of blocks by `sort(unique(blocks))`; columns should be in the order of `conditions`, if specified. (optional)

`block_prob` Use for a two-arm design in which the probability of assignment to treatment varies across blocks. Must be in the same order as `sort(unique(blocks))`. (optional)

`block_prob_each` Use for a multi-arm design in which assignment probabilities vary across blocks. Must be a matrix with one row per block and one column per treatment arm. Each row must sum to 1. Rows respect the ordering of `sort(unique(blocks))`. (optional)

`num_arms` The number of treatment arms. If unspecified, determined from the other arguments. (optional)

`conditions` A character vector giving the names of the treatment groups. If unspecified, the treatment groups will be named 0 (for control) and 1 (for treatment) in a two-arm trial and T1, T2, T3, in a multi-arm trial. A two-group design in which `num_arms` is set to 2 will use condition names T1 and T2. (optional)

`check_inputs` Logical. Whether to verify before assigning that the arguments are internally consistent: that counts sum to the block sizes, that probabilities lie between 0 and 1 and sum to 1, that matrices have one row per block, and so on. Defaults to `TRUE`. `FALSE` skips the checking only: `num_arms` and `conditions` are still derived from the other arguments, so the same call draws the same assignment either way. What goes is the verification, and an impossible design is then no longer refused. `block_m` larger than a block, for instance, quietly treats the whole block. Declaring the design once with `declare_ra()` and drawing from it with `conduct_ra()` is the usual way to avoid re-checking the same arguments in a simulation. (optional)

`.block_int` Internal use only. Pre-computed integer encoding of `blocks`, passed by `conduct_ra()` when a declaration was created with `declare_ra()`. Users should never set this argument. (optional)

`.N_per_block` Internal use only. Pre-computed block sizes corresponding to `.block_int`, passed by `conduct_ra()`. Users should never set this argument. (optional)

Details

In the simplest two-arm case with no arguments beyond `blocks`, the function assigns approximately half the units in each block to treatment. Researchers can specify exact counts (via `block_m`) or target probabilities that are held constant (via `prob`) or allowed to vary (via `block_prob`) across blocks.

Value

A vector of length N indicating the treatment condition of each unit. Numeric in a two-arm trial; a factor (ordered by `conditions`) in a multi-arm trial.

See Also

complete_ra(), block_and_cluster_ra(), strata_rs(), block_ra_probabilities()

Examples

```
# Two-arm Designs

blocks <- rep(c("A", "B","C"), times = c(50, 100, 200))
Z <- block_ra(blocks = blocks)
table(blocks, Z)

Z <- block_ra(blocks = blocks, prob = 0.3)
table(blocks, Z)

Z <- block_ra(blocks = blocks, block_prob = c(0.1, 0.2, 0.3))
table(blocks, Z)

Z <- block_ra(blocks = blocks,
              prob_unit = rep(c(0.1, 0.2, 0.3),
                              times = c(50, 100, 200)))
table(blocks, Z)

Z <- block_ra(blocks = blocks, m = 20)
table(blocks, Z)

Z <- block_ra(blocks = blocks, block_m = c(20, 30, 40))
table(blocks, Z)

Z <- block_ra(blocks = blocks,
              m_unit = rep(c(20, 30, 40),
                           times = c(50, 100, 200)))
table(blocks, Z)

block_m_each <- rbind(c(25, 25),
                 c(50, 50),
                 c(100, 100))

Z <- block_ra(blocks = blocks, block_m_each = block_m_each)
table(blocks, Z)

block_m_each <- rbind(c(10, 40),
                 c(30, 70),
                 c(50, 150))

Z <- block_ra(blocks = blocks, block_m_each = block_m_each,
              conditions = c("control", "treatment"))
table(blocks, Z)

# Multi-arm Designs
Z <- block_ra(blocks = blocks, num_arms = 3)
table(blocks, Z)
```

```
block_m_each <- rbind(c(10, 20, 20),
                      c(30, 50, 20),
                      c(50, 75, 75))
Z <- block_ra(blocks = blocks, block_m_each = block_m_each)
table(blocks, Z)

Z <- block_ra(blocks = blocks, block_m_each = block_m_each,
              conditions = c("control", "placebo", "treatment"))
table(blocks, Z)

Z <- block_ra(blocks = blocks, prob_each = c(0.1, 0.1, 0.8))
table(blocks, Z)
```

`block_ra_probabilities` *Probabilities of assignment: Block Random Assignment*

Description

Returns the probability that each unit is assigned to each condition under block random assignment. Units in different blocks routinely have different probabilities, which is exactly when these numbers are needed.

Usage

```
block_ra_probabilities(
  blocks = NULL,
  prob = NULL,
  prob_unit = NULL,
  prob_each = NULL,
  m = NULL,
  m_unit = NULL,
  block_m = NULL,
  block_m_each = NULL,
  block_prob = NULL,
  block_prob_each = NULL,
  num_arms = NULL,
  conditions = NULL,
  check_inputs = TRUE
)
```

Arguments

`blocks`	A vector of length N indicating which block each unit belongs to. Can be character, factor, or numeric. (required)

prob — Use for a two-arm design in which either floor(N_block*prob) or ceiling(N_block*prob) units are assigned to treatment within each block. Which of the two is used is itself random: the ceiling is drawn with probability equal to the fractional part of N_block*prob and the floor otherwise, which makes each unit's probability of assignment exactly prob. When N_block*prob is a whole number the count is fixed. Must be a real number between 0 and 1. (optional)

prob_unit — Use for a two-arm design. Must be of length N. tapply(prob_unit, blocks, unique) will be passed to block_prob. (optional)

prob_each — Use for a multi-arm design in which the values of prob_each determine the probabilities of assignment to each treatment condition. Must be a numeric vector giving the probability of assignment to each condition. All entries must be nonnegative real numbers between 0 and 1 and the total must sum to 1. Because of integer rounding, the exact number of units assigned to each condition may differ slightly from assignment to assignment, but the overall probability of assignment is exactly prob_each. (optional)

m — Use for a two-arm design in which the scalar m gives the fixed number of units to assign to treatment within every block. This count does not vary across blocks. (optional)

m_unit — Use for a two-arm design. Must be of length N. tapply(m_unit, blocks, unique) will be passed to block_m. (optional)

block_m — Use for a two-arm design in which block_m gives the number of units to assign to treatment within each block. Must be a numeric vector as long as the number of blocks, in the same order as sort(unique(blocks)). (optional)

block_m_each — Use for a multi-arm design in which block_m_each gives the number of units assigned to each condition within each block. Must be a matrix with one row per block and one column per treatment arm. Rows should respect the ordering of blocks by sort(unique(blocks)); columns should be in the order of conditions, if specified. (optional)

block_prob — Use for a two-arm design in which the probability of assignment to treatment varies across blocks. Must be in the same order as sort(unique(blocks)). (optional)

block_prob_each — Use for a multi-arm design in which assignment probabilities vary across blocks. Must be a matrix with one row per block and one column per treatment arm. Each row must sum to 1. Rows respect the ordering of sort(unique(blocks)). (optional)

num_arms — The number of treatment arms. If unspecified, determined from the other arguments. (optional)

conditions — A character vector giving the names of the treatment groups. If unspecified, the treatment groups will be named 0 (for control) and 1 (for treatment) in a two-arm trial and T1, T2, T3, in a multi-arm trial. A two-group design in which num_arms is set to 2 will use condition names T1 and T2. (optional)

check_inputs — Logical. Whether to verify before assigning that the arguments are internally consistent: that counts sum to the block sizes, that probabilities lie between 0 and 1 and sum to 1, that matrices have one row per block, and so on. Defaults

to TRUE. FALSE skips the checking only: num_arms and conditions are still derived from the other arguments, so the same call draws the same assignment either way. What goes is the verification, and an impossible design is then no longer refused. block_m larger than a block, for instance, quietly treats the whole block. Declaring the design once with declare_ra() and drawing from it with conduct_ra() is the usual way to avoid re-checking the same arguments in a simulation. (optional)

Details

These are the quantities inverse-probability weights are built from: weight each unit by the reciprocal of the probability of the condition it landed in, which obtain_condition_probabilities() extracts for you.

Value

A matrix with N rows and one column per treatment condition, with columns named prob_<condition>. Entry (i, j) is the probability that unit i is assigned to condition j, and every row sums to 1.

See Also

block_ra()

Examples

```
blocks <- rep(c("A", "B","C"), times = c(50, 100, 200))
prob_mat <- block_ra_probabilities(blocks = blocks)
head(prob_mat)

prob_mat <- block_ra_probabilities(blocks = blocks, m = 20)
head(prob_mat)

block_m_each <- rbind(c(25, 25),
                c(50, 50),
                c(100, 100))

prob_mat <- block_ra_probabilities(blocks = blocks, block_m_each = block_m_each)
head(prob_mat)

block_m_each <- rbind(c(10, 40),
                c(30, 70),
                c(50, 150))

prob_mat <- block_ra_probabilities(blocks = blocks,
                                   block_m_each = block_m_each,
                                   conditions = c("control", "treatment"))
head(prob_mat)

prob_mat <- block_ra_probabilities(blocks = blocks, num_arms = 3)
head(prob_mat)

block_m_each <- rbind(c(10, 20, 20),
```

```
                c(30, 50, 20),
                c(50, 75, 75))
prob_mat <- block_ra_probabilities(blocks = blocks, block_m_each = block_m_each)
head(prob_mat)

prob_mat <- block_ra_probabilities(blocks = blocks, block_m_each = block_m_each,
                       conditions = c("control", "placebo", "treatment"))
head(prob_mat)

prob_mat <- block_ra_probabilities(blocks = blocks, prob_each = c(0.1, 0.1, 0.8))
head(prob_mat)
```

`cluster_ra` *Cluster Random Assignment*

Description

`cluster_ra` assigns entire groups of units (clusters) to treatment conditions, so that all units within a cluster share the same treatment status. Cluster assignment is appropriate when the intervention can only be delivered at the group level (for example, a school-wide program that cannot be withheld from individual students), when spillovers within groups make individual-level assignment infeasible, or when the treatment is itself defined as a group-level condition. Because all units in a cluster move together, the effective sample size for estimating average effects is the number of clusters, not the number of units. Clustering therefore typically increases sampling variability relative to complete or block random assignment; the precision loss grows with the intra-cluster correlation in potential outcomes.

Usage

```
cluster_ra(
  clusters = NULL,
  m = NULL,
  m_unit = NULL,
  m_each = NULL,
  prob = NULL,
  prob_unit = NULL,
  prob_each = NULL,
  num_arms = NULL,
  conditions = NULL,
  simple = FALSE,
  check_inputs = TRUE
)
```

Arguments

`clusters`	A vector of length N indicating which cluster each unit belongs to. (required)

`m` Use for a two-arm design in which exactly `m` clusters are assigned to treatment. (optional)

`m_unit` Use for a two-arm design. `unique(m_unit)` clusters are assigned to treatment; must be the same for all units and of length N. (optional)

`m_each` Use for a multi-arm design. A numeric vector giving the number of clusters assigned to each condition; must sum to the total number of clusters. (optional)

`prob` Use for a two-arm design in which either `floor(N_clusters*prob)` or `ceiling(N_clusters*prob)` clusters are assigned to treatment. Which of the two is used is itself random: the ceiling is drawn with probability equal to the fractional part of `N_clusters*prob` and the floor otherwise, so that each cluster's probability of assignment is exactly `prob`. When `N_clusters*prob` is a whole number the count is fixed. Must be between 0 and 1. (optional)

`prob_unit` Use for a two-arm design. `unique(prob_unit)` will be passed to the `prob` argument and must be the same for all units. (optional)

`prob_each` Use for a multi-arm design. A numeric vector giving the probability of assignment to each condition; entries must be nonnegative, sum to 1. Because of integer rounding, the exact number of clusters assigned to each condition may differ slightly from assignment to assignment, but the overall probability of assignment is exactly `prob_each`. (optional)

`num_arms` The total number of treatment arms. If unspecified, determined from `m_each` or `conditions`. (optional)

`conditions` A character vector giving the names of the treatment groups. If unspecified, groups will be named T1, T2, T3, etc. (optional)

`simple` Logical, defaults to `FALSE`. If `TRUE`, clusters are assigned to conditions independently (simple random assignment at the cluster level), so the number of treated clusters varies from draw to draw. Do not specify `m` or `m_each` when `simple = TRUE`. (optional)

`check_inputs` Logical. Whether to verify before assigning that the arguments are internally consistent: that counts sum to the number of clusters, that probabilities lie between 0 and 1 and sum to 1, and so on. Defaults to `TRUE`. `FALSE` skips the checking only: `num_arms` and `conditions` are still derived from the other arguments, so the same call draws the same assignment either way. What goes is the verification, and an impossible design is then no longer refused. `block_m` larger than a block, for instance, quietly treats the whole block. Declaring the design once with `declare_ra()` and drawing from it with `conduct_ra()` is the usual way to avoid re-checking the same arguments in a simulation. (optional)

Details

By default, `cluster_ra` conducts complete random assignment at the cluster level: a fixed number of clusters are assigned to each condition on every draw. Setting `simple = TRUE` switches to independent Bernoulli assignment of clusters.

Value

A vector of length N indicating the treatment condition of each unit. Every unit in a cluster receives the same value. Numeric in a two-arm trial; a factor (ordered by `conditions`) in a multi-arm trial.

See Also

complete_ra(), block_and_cluster_ra(), cluster_rs(), cluster_ra_probabilities()

Examples

```
# Ten clusters, of sizes 1 through 10
clusters <- rep(letters[1:10], times = 1:10)

# Two Group Designs

Z <- cluster_ra(clusters = clusters)
table(Z, clusters)

Z <- cluster_ra(clusters = clusters, m = 4)
table(Z, clusters)

Z <- cluster_ra(clusters = clusters, m_each = c(6, 4),
                conditions = c("control", "treatment"))
table(Z, clusters)

# Multi-arm Designs
Z <- cluster_ra(clusters = clusters, num_arms = 3)
table(Z, clusters)

Z <- cluster_ra(clusters = clusters, m_each = c(3, 3, 4))
table(Z, clusters)

Z <- cluster_ra(clusters = clusters, m_each = c(3, 3, 4),
                conditions = c("control", "placebo", "treatment"))
table(Z, clusters)

Z <- cluster_ra(clusters = clusters,
                conditions = c("control", "placebo", "treatment"))
table(Z, clusters)
```

---

cluster_ra_probabilities

Probabilities of assignment: Cluster Random Assignment

Description

Returns the probability that each unit is assigned to each condition under cluster random assignment. Every unit in a cluster shares its cluster's probability, since clusters move together.

Usage

```
cluster_ra_probabilities(
  clusters = NULL,
  m = NULL,
```

```
  m_unit = NULL,
  m_each = NULL,
  prob = NULL,
  prob_unit = NULL,
  prob_each = NULL,
  num_arms = NULL,
  conditions = NULL,
  simple = FALSE,
  check_inputs = TRUE
)
```

Arguments

clusters	A vector of length N indicating which cluster each unit belongs to. (required)
m	Use for a two-arm design in which exactly m clusters are assigned to treatment. (optional)
m_unit	Use for a two-arm design. unique(m_unit) clusters are assigned to treatment; must be the same for all units and of length N. (optional)
m_each	Use for a multi-arm design. A numeric vector giving the number of clusters assigned to each condition; must sum to the total number of clusters. (optional)
prob	Use for a two-arm design in which either floor(N_clusters*prob) or ceiling(N_clusters*prob) clusters are assigned to treatment. Which of the two is used is itself random: the ceiling is drawn with probability equal to the fractional part of N_clusters*prob and the floor otherwise, so that each cluster's probability of assignment is exactly prob. When N_clusters*prob is a whole number the count is fixed. Must be between 0 and 1. (optional)
prob_unit	Use for a two-arm design. unique(prob_unit) will be passed to the prob argument and must be the same for all units. (optional)
prob_each	Use for a multi-arm design. A numeric vector giving the probability of assignment to each condition; entries must be nonnegative, sum to 1. Because of integer rounding, the exact number of clusters assigned to each condition may differ slightly from assignment to assignment, but the overall probability of assignment is exactly prob_each. (optional)
num_arms	The total number of treatment arms. If unspecified, determined from m_each or conditions. (optional)
conditions	A character vector giving the names of the treatment groups. If unspecified, groups will be named T1, T2, T3, etc. (optional)
simple	Logical, defaults to FALSE. If TRUE, clusters are assigned to conditions independently (simple random assignment at the cluster level), so the number of treated clusters varies from draw to draw. Do not specify m or m_each when simple = TRUE. (optional)
check_inputs	Logical. Whether to verify before assigning that the arguments are internally consistent: that counts sum to the number of clusters, that probabilities lie between 0 and 1 and sum to 1, and so on. Defaults to TRUE. FALSE skips the checking only: num_arms and conditions are still derived from the other arguments, so the same call draws the same assignment either way. What goes is the

verification, and an impossible design is then no longer refused. `block_m` larger than a block, for instance, quietly treats the whole block. Declaring the design once with `declare_ra()` and drawing from it with `conduct_ra()` is the usual way to avoid re-checking the same arguments in a simulation. (optional)

Details

These are the quantities inverse-probability weights are built from: weight each unit by the reciprocal of the probability of the condition it landed in, which `obtain_condition_probabilities()` extracts for you.

Value

A matrix with N rows and one column per treatment condition, with columns named `prob_<condition>`. Entry (i, j) is the probability that unit i is assigned to condition j, and every row sums to 1.

See Also

`cluster_ra()`

Examples

```
# Two Group Designs
clusters <- rep(letters[1:10], times = 1:10)
prob_mat <- cluster_ra_probabilities(clusters = clusters)
head(prob_mat)

prob_mat <- cluster_ra_probabilities(clusters = clusters, m = 4)
head(prob_mat)

prob_mat <- cluster_ra_probabilities(clusters = clusters,
                                     m_each = c(6, 4),
                                     conditions = c("control", "treatment"))

# Multi-arm Designs
prob_mat <- cluster_ra_probabilities(clusters = clusters, num_arms = 3)
head(prob_mat)

prob_mat <- cluster_ra_probabilities(clusters = clusters, m_each = c(3, 3, 4))
head(prob_mat)

prob_mat <- cluster_ra_probabilities(clusters = clusters, m_each = c(3, 3, 4),
                          conditions = c("control", "placebo", "treatment"))
head(prob_mat)

prob_mat <- cluster_ra_probabilities(clusters = clusters,
                          conditions = c("control", "placebo", "treatment"))
head(prob_mat)

prob_mat <- cluster_ra_probabilities(clusters = clusters,
                                     prob_each = c(0.1, 0.2, 0.7))
head(prob_mat)
```

`cluster_rs` *Cluster Random Sampling*

Description

`cluster_rs` draws whole groups of units (clusters) into the sample, so that either every unit in a cluster is sampled or none of them is. Use it when the sampling frame lists groups rather than individuals, for example when villages are drawn and then everyone in the drawn villages is interviewed. Because units come in whole clusters, the effective sample size is closer to the number of clusters than to the number of units.

Usage

```
cluster_rs(
  clusters = NULL,
  n = NULL,
  n_unit = NULL,
  prob = NULL,
  prob_unit = NULL,
  simple = FALSE,
  check_inputs = TRUE
)
```

Arguments

`clusters`	A vector of length N indicating which cluster each unit belongs to. (required)
`n`	Use for a design in which exactly `n` clusters are sampled. (optional)
`n_unit`	`unique(n_unit)` will be passed to `n`; must be the same for all units and of length N. (optional)
`prob`	Use for a design in which either `floor(N_clusters*prob)` or `ceiling(N_clusters*prob)` clusters are sampled. Which of the two is used is itself random: the ceiling is drawn with probability equal to the fractional part of `N_clusters*prob` and the floor otherwise, which makes each cluster's probability of inclusion exactly `prob`. Must be a real number between 0 and 1 inclusive. (optional)
`prob_unit`	`unique(prob_unit)` will be passed to `prob`; must be the same for all units and of length N. (optional)
`simple`	Logical, defaults to `FALSE`. If `TRUE`, clusters are drawn independently (simple random sampling of clusters), so the number of sampled clusters varies from draw to draw. Do not specify `n` when `simple = TRUE`. (optional)

check_inputs — Logical. Whether to verify before sampling that the arguments are internally consistent: that n does not exceed the number of clusters, that probabilities lie between 0 and 1, and so on. Defaults to TRUE. Set to FALSE to skip the checks when drawing many samples from arguments that have already been verified; declaring the design once with declare_rs() and drawing from it with draw_rs() does this for you. (optional)

Details

By default the clusters are drawn by complete random sampling, so a fixed number of clusters is sampled on every draw. Setting simple = TRUE draws each cluster independently instead, using simple_rs().

Value

A numeric vector of length N indicating whether each unit is sampled (1) or not (0). Every unit in a cluster receives the same value.

See Also

complete_rs(), strata_and_cluster_rs(), cluster_ra(), cluster_rs_probabilities()

Examples

```
# Ten clusters, of sizes 1 through 10
clusters <- rep(letters[1:10], times = 1:10)

S <- cluster_rs(clusters = clusters)
table(S, clusters)

S <- cluster_rs(clusters = clusters, n = 4)
table(S, clusters)

# Each cluster drawn independently, so the number sampled varies
S <- cluster_rs(clusters = clusters, prob = 0.4, simple = TRUE)
table(S, clusters)
```

---

cluster_rs_probabilities

Inclusion probabilities: Cluster Sampling

Description

Returns each unit's probability of being sampled when whole clusters are drawn. Every unit in a cluster shares its cluster's probability.

Usage

```
cluster_rs_probabilities(
  clusters = NULL,
  n = NULL,
  n_unit = NULL,
  prob = NULL,
  prob_unit = NULL,
  simple = FALSE,
  check_inputs = TRUE
)
```

Arguments

clusters	A vector of length N indicating which cluster each unit belongs to. (required)
n	Use for a design in which exactly n clusters are sampled. (optional)
n_unit	unique(n_unit) will be passed to n; must be the same for all units and of length N. (optional)
prob	Use for a design in which either floor(N_clusters*prob) or ceiling(N_clusters*prob) clusters are sampled. Which of the two is used is itself random: the ceiling is drawn with probability equal to the fractional part of N_clusters*prob and the floor otherwise, which makes each cluster's probability of inclusion exactly prob. Must be a real number between 0 and 1 inclusive. (optional)
prob_unit	unique(prob_unit) will be passed to prob; must be the same for all units and of length N. (optional)
simple	Logical, defaults to FALSE. If TRUE, clusters are drawn independently (simple random sampling of clusters), so the number of sampled clusters varies from draw to draw. Do not specify n when simple = TRUE. (optional)
check_inputs	Logical. Whether to verify before sampling that the arguments are internally consistent: that n does not exceed the number of clusters, that probabilities lie between 0 and 1, and so on. Defaults to TRUE. Set to FALSE to skip the checks when drawing many samples from arguments that have already been verified; declaring the design once with declare_rs() and drawing from it with draw_rs() does this for you. (optional)

Details

These are the quantities inverse-probability weights are built from: weight each sampled unit by the reciprocal of its inclusion probability, which obtain_inclusion_probabilities() extracts for you.

Value

A numeric vector of length N giving each unit's probability of being included in the sample. Every unit in a cluster shares one probability.

See Also

cluster_rs()

Examples

```
clusters <- rep(letters[1:10], times = 1:10)

probs <- cluster_rs_probabilities(clusters = clusters)
table(probs, clusters)

probs <- cluster_rs_probabilities(clusters = clusters, n = 4)
table(probs, clusters)

probs <- cluster_rs_probabilities(clusters = clusters, prob = 0.3)
table(probs, clusters)
```

---

complete_ra *Complete Random Assignment*

Description

`complete_ra` assigns exactly fixed numbers of units to each treatment condition. In the canonical two-arm case, exactly `m` of `N` units are assigned to treatment and `N-m` to control on every draw. This guarantee that the counts are fixed is the defining feature of complete random assignment, and it is what distinguishes it from simple random assignment (where counts vary from draw to draw).

Usage

```
complete_ra(
  N,
  m = NULL,
  m_unit = NULL,
  m_each = NULL,
  prob = NULL,
  prob_unit = NULL,
  prob_each = NULL,
  num_arms = NULL,
  conditions = NULL,
  check_inputs = TRUE
)
```

Arguments

`N`	The number of units. Must be a positive integer. (required)
`m`	Use for a two-arm design: exactly `m` units are assigned to treatment and `N-m` to control. (optional)
`m_unit`	Use for a two-arm design. `unique(m_unit)` units are assigned to treatment; must be the same for all units and of length N. (optional)

m_each	Use for a multi-arm design. A numeric vector giving the exact number of units assigned to each condition; must sum to N. (optional)
prob	Use for a two-arm design: either floor(N*prob) or ceiling(N*prob) units are assigned to treatment so that the marginal probability of assignment equals exactly prob. Must be between 0 and 1. One edge is deliberate: when ceiling(N*prob) == N (for instance N = 3, prob = 0.9), exactly floor(N*prob) units are treated, never all N, so the marginal probability is floor(N*prob)/N; complete_ra_probabilities() reports the probability actually used. (optional)
prob_unit	Use for a two-arm design. unique(prob_unit) will be passed to the prob argument; must be the same for all units. (optional)
prob_each	Use for a multi-arm design. A numeric vector giving the probability of assignment to each condition; entries must be nonnegative and sum to 1. Due to integer rounding the exact count assigned to each condition may differ slightly from draw to draw, but the overall probability of assignment is exactly prob_each. (optional)
num_arms	The number of treatment arms. If unspecified, determined from the other arguments. (optional)
conditions	A character vector giving the names of the treatment groups. If unspecified, groups will be named 0 and 1 in a two-arm trial and T1, T2, T3, in a multi-arm trial. A two-group design in which num_arms is set to 2 will use condition names T1 and T2. (optional)
check_inputs	Logical. Whether to verify before assigning that the arguments are internally consistent: that counts sum to N, that probabilities lie between 0 and 1 and sum to 1, that vectors are of length N, and so on. Defaults to TRUE. FALSE skips the checking only: num_arms and conditions are still derived from the other arguments, so the same call draws the same assignment either way. What goes is the verification, and an impossible design is then no longer refused. block_m larger than a block, for instance, quietly treats the whole block. Declaring the design once with declare_ra() and drawing from it with conduct_ra() is the usual way to avoid re-checking the same arguments in a simulation. (optional)

Details

Researchers can specify counts directly (via m or m_each) or target probabilities (via prob or prob_each). When probabilities are specified and the implied counts are not integers, complete_ra uses stochastic rounding to ensure that the overall probability of assignment exactly equals the target. In a two-arm design, either floor(N*prob) or ceiling(N*prob) units are assigned to treatment, with the draw between these two values chosen so that Pr(treatment) equals exactly prob. In a multi-arm design, the remaining units after floor allocation are assigned using a single round of simple random assignment calibrated to hit the exact target probabilities.

If only N is specified, a balanced two-arm trial (prob = 0.5) is assumed. When N is odd, either floor(N/2) or ceiling(N/2) units are assigned to treatment.

Value

A vector of length N indicating the treatment condition of each unit. Numeric in a two-arm trial; a factor (ordered by conditions) in a multi-arm trial.

See Also

simple_ra(), block_ra(), cluster_ra(), complete_rs(), complete_ra_probabilities()

Examples

```
# Two-arm Designs
Z <- complete_ra(N = 100)
table(Z)

Z <- complete_ra(N = 100, m = 50)
table(Z)

Z <- complete_ra(N = 100, m_unit = rep(30, 100))
table(Z)

Z <- complete_ra(N = 100, prob = 0.111)
table(Z)

Z <- complete_ra(N = 100, prob_unit = rep(0.1, 100))
table(Z)

Z <- complete_ra(N = 100, conditions = c("control", "treatment"))
table(Z)


# Multi-arm Designs
Z <- complete_ra(N = 100, num_arms = 3)
table(Z)

Z <- complete_ra(N = 100, m_each = c(30, 30, 40))
table(Z)

Z <- complete_ra(N = 100, prob_each = c(0.1, 0.2, 0.7))
table(Z)

Z <- complete_ra(N = 100, conditions = c("control", "placebo", "treatment"))
table(Z)

# Special Cases
# Two-arm trial where the conditions are by default "T1" and "T2"
Z <- complete_ra(N = 100, num_arms = 2)
table(Z)

# If N = m, every unit is assigned to treatment with probability 1
complete_ra(N = 2, m = 2)

# The single-unit case works the same way: m = 1 out of N = 1 is treated
# with probability 1. Up through randomizr 0.12.0 this case was instead
# treated as a coin flip, so the unit was assigned to treatment only half of
# the time. The change is noted here because it silently alters the
# probabilities of assignment in code written against those versions.
complete_ra(N = 1, m = 1)
```

`complete_ra_probabilities` *Probabilities of assignment: Complete Random Assignment*

Description

Returns the probability that each unit is assigned to each condition under complete random assignment. When the implied counts are not integers the probabilities account for the stochastic rounding `complete_ra` uses, so they equal the target exactly rather than approximately.

Usage

```
complete_ra_probabilities(
  N,
  m = NULL,
  m_unit = NULL,
  m_each = NULL,
  prob = NULL,
  prob_unit = NULL,
  prob_each = NULL,
  num_arms = NULL,
  conditions = NULL,
  check_inputs = TRUE
)
```

Arguments

`N`	The number of units. Must be a positive integer. (required)
`m`	Use for a two-arm design: exactly `m` units are assigned to treatment and `N-m` to control. (optional)
`m_unit`	Use for a two-arm design. `unique(m_unit)` units are assigned to treatment; must be the same for all units and of length N. (optional)
`m_each`	Use for a multi-arm design. A numeric vector giving the exact number of units assigned to each condition; must sum to N. (optional)
`prob`	Use for a two-arm design: either `floor(N*prob)` or `ceiling(N*prob)` units are assigned to treatment so that the marginal probability of assignment equals exactly `prob`. Must be between 0 and 1. One edge is deliberate: when `ceiling(N*prob) == N` (for instance `N = 3, prob = 0.9`), exactly `floor(N*prob)` units are treated, never all `N`, so the marginal probability is `floor(N*prob)/N`; `complete_ra_probabilities()` reports the probability actually used. (optional)
`prob_unit`	Use for a two-arm design. `unique(prob_unit)` will be passed to the `prob` argument; must be the same for all units. (optional)

`prob_each` Use for a multi-arm design. A numeric vector giving the probability of assignment to each condition; entries must be nonnegative and sum to 1. Due to integer rounding the exact count assigned to each condition may differ slightly from draw to draw, but the overall probability of assignment is exactly `prob_each`. (optional)

`num_arms` The number of treatment arms. If unspecified, determined from the other arguments. (optional)

`conditions` A character vector giving the names of the treatment groups. If unspecified, groups will be named 0 and 1 in a two-arm trial and T1, T2, T3, in a multi-arm trial. A two-group design in which `num_arms` is set to 2 will use condition names T1 and T2. (optional)

`check_inputs` Logical. Whether to verify before assigning that the arguments are internally consistent: that counts sum to N, that probabilities lie between 0 and 1 and sum to 1, that vectors are of length N, and so on. Defaults to `TRUE`. `FALSE` skips the checking only: `num_arms` and `conditions` are still derived from the other arguments, so the same call draws the same assignment either way. What goes is the verification, and an impossible design is then no longer refused. `block_m` larger than a block, for instance, quietly treats the whole block. Declaring the design once with `declare_ra()` and drawing from it with `conduct_ra()` is the usual way to avoid re-checking the same arguments in a simulation. (optional)

Details

These are the quantities inverse-probability weights are built from: weight each unit by the reciprocal of the probability of the condition it landed in, which `obtain_condition_probabilities()` extracts for you.

Value

A matrix with N rows and one column per treatment condition, with columns named `prob_<condition>`. Entry (i, j) is the probability that unit i is assigned to condition j, and every row sums to 1.

See Also

`complete_ra()`

Examples

```
# 2-arm designs
prob_mat <- complete_ra_probabilities(N = 100)
head(prob_mat)

prob_mat <- complete_ra_probabilities(N = 100, m = 50)
head(prob_mat)

prob_mat <- complete_ra_probabilities(N = 100, prob = 0.3)
head(prob_mat)

prob_mat <- complete_ra_probabilities(N = 100, m_each = c(30, 70),
```

```
                          conditions = c("control", "treatment"))
head(prob_mat)

# Multi-arm Designs
prob_mat <- complete_ra_probabilities(N = 100, num_arms = 3)
head(prob_mat)

prob_mat <- complete_ra_probabilities(N = 100, m_each = c(30, 30, 40))
head(prob_mat)

prob_mat <- complete_ra_probabilities(N = 100, m_each = c(30, 30, 40),
                          conditions = c("control", "placebo", "treatment"))
head(prob_mat)

prob_mat <- complete_ra_probabilities(N = 100, conditions = c("control", "placebo", "treatment"))
head(prob_mat)

prob_mat <- complete_ra_probabilities(N = 100, prob_each = c(0.2, 0.7, 0.1))
head(prob_mat)
```

`complete_rs` *Complete Random Sampling*

Description

`complete_rs` draws a sample of a fixed size: exactly `n` of `N` units are sampled on every draw. Fixing the sample size is what distinguishes it from simple random sampling, where the realized size varies.

Usage

```
complete_rs(
  N,
  n = NULL,
  n_unit = NULL,
  prob = NULL,
  prob_unit = NULL,
  check_inputs = TRUE
)
```

Arguments

`N`	The number of units in the sampling frame. Must be a positive integer. (required)
`n`	Use for a design in which exactly `n` units are sampled. (optional)
`n_unit`	`unique(n_unit)` will be passed to `n`; must be the same for all units and of length N. (optional)

prob	Use for a design in which either floor(N*prob) or ceiling(N*prob) units are sampled, chosen so that each unit's probability of inclusion is exactly prob. Must be a real number between 0 and 1 inclusive. (optional)
prob_unit	unique(prob_unit) will be passed to prob; must be the same for all units and of length N. Under complete random sampling the probability cannot vary by unit; use simple_rs() if it must. (optional)
check_inputs	Logical. Whether to verify before sampling that the arguments are internally consistent: that n does not exceed N, that probabilities lie between 0 and 1, that vectors are of length N, and so on. Defaults to TRUE. Set to FALSE to skip the checks when drawing many samples from arguments that have already been verified; declaring the design once with declare_rs() and drawing from it with draw_rs() does this for you. (optional)

Details

Set the number of units to sample directly with n, or give a target probability with prob and let complete_rs work out the number. When N*prob is not a whole number, either floor(N*prob) or ceiling(N*prob) units are sampled: the ceiling is drawn with probability equal to the fractional part of N*prob and the floor otherwise, which makes each unit's probability of inclusion exactly prob. Specify N and not more than one of n or prob.

If only N is specified, half the units are sampled. When N is odd, either floor(N/2) or ceiling(N/2) units are sampled.

Value

A numeric vector of length N indicating whether each unit is sampled (1) or not (0).

See Also

simple_rs(), strata_rs(), cluster_rs(), complete_ra(), complete_rs_probabilities()

Examples

```
S <- complete_rs(N = 100)
table(S)

S <- complete_rs(N = 100, n = 50)
table(S)

S <- complete_rs(N = 100, n_unit = rep(30, 100))
table(S)

S <- complete_rs(N = 100, prob = 0.111)
table(S)

S <- complete_rs(N = 100, prob_unit = rep(0.1, 100))
table(S)

# If N = n, every unit is sampled with probability 1
complete_rs(N = 2, n = 2)
```

```
# The single-unit case works the same way: n = 1 out of N = 1 is sampled
# with probability 1. Up through randomizr 0.12.0 this case was instead
# treated as a coin flip, so the unit was sampled only half of the time.
# The change is noted here because it silently alters the inclusion
# probabilities in code written against those versions.
complete_rs(N = 1, n = 1)
```

`complete_rs_probabilities` *Inclusion probabilities: Complete Random Sampling*

Description

Returns each unit's probability of being sampled under complete random sampling, where the sample size is fixed on every draw.

Usage

```
complete_rs_probabilities(
  N,
  n = NULL,
  n_unit = NULL,
  prob = NULL,
  prob_unit = NULL,
  check_inputs = TRUE
)
```

Arguments

`N`	The number of units in the sampling frame. Must be a positive integer. (required)
`n`	Use for a design in which exactly `n` units are sampled. (optional)
`n_unit`	`unique(n_unit)` will be passed to `n`; must be the same for all units and of length N. (optional)
`prob`	Use for a design in which either `floor(N*prob)` or `ceiling(N*prob)` units are sampled, chosen so that each unit's probability of inclusion is exactly `prob`. Must be a real number between 0 and 1 inclusive. (optional)
`prob_unit`	`unique(prob_unit)` will be passed to `prob`; must be the same for all units and of length N. Under complete random sampling the probability cannot vary by unit; use `simple_rs()` if it must. (optional)
`check_inputs`	Logical. Whether to verify before sampling that the arguments are internally consistent: that `n` does not exceed N, that probabilities lie between 0 and 1, that vectors are of length N, and so on. Defaults to `TRUE`. Set to `FALSE` to skip

the checks when drawing many samples from arguments that have already been verified; declaring the design once with `declare_rs()` and drawing from it with `draw_rs()` does this for you. (optional)

Details

These are the quantities inverse-probability weights are built from: weight each sampled unit by the reciprocal of its inclusion probability, which `obtain_inclusion_probabilities()` extracts for you.

Value

A numeric vector of length N giving each unit's probability of being included in the sample.

See Also

`complete_rs()`

Examples

```
probs <- complete_rs_probabilities(N = 100)
table(probs)

probs <- complete_rs_probabilities(N = 100, n = 50)
table(probs)

probs <- complete_rs_probabilities(N = 100, prob = 0.3)
table(probs)
```

---

conduct_ra *Conduct a Random Assignment*

Description

`conduct_ra` draws one random assignment from a design. Give it a declaration made by `declare_ra()`, or describe the design inline with the same arguments `declare_ra()` takes. Declaring first pays off when the same design is drawn repeatedly, or when the assignment probabilities are needed later by `obtain_condition_probabilities()`.

Usage

```
conduct_ra(
  declaration = NULL,
  N = NULL,
  blocks = NULL,
  clusters = NULL,
  m = NULL,
  m_unit = NULL,
```

```
  m_each = NULL,
  prob = NULL,
  prob_unit = NULL,
  prob_each = NULL,
  prob_unit_each = NULL,
  block_m = NULL,
  block_m_each = NULL,
  block_prob = NULL,
  block_prob_each = NULL,
  num_arms = NULL,
  conditions = NULL,
  simple = FALSE,
  ra_type = NULL,
  formula = NULL,
  permutation_matrix = NULL,
  check_inputs = TRUE,
  data = NULL
)
```

Arguments

declaration	A random assignment declaration, created by declare_ra(). Supply either a declaration or the design arguments listed below, which are the ones declare_ra() takes: given those, conduct_ra builds a declaration internally and draws one assignment from it. (optional)
N	The number of units. A positive integer. Optional when data, formula, or the length of prob_unit (or blocks, or clusters) identifies N.
blocks	A vector of length N indicating which block each unit belongs to, or, when data is supplied, the name of the column holding it. Supply to use blocked random assignment. (optional)
clusters	A vector of length N indicating which cluster each unit belongs to, or, when data is supplied, the name of the column holding it. Supply to use cluster random assignment. (optional)
m	Use for a two-arm design: exactly m units (or clusters) are assigned to treatment. In a blocked design, exactly m units in each block are treated. (optional)
m_unit	Use for a two-arm trial. A vector of length N; a single number is refused, since that is what m is for. Under complete random assignment, must be constant across units. Under blocked random assignment, must be constant within blocks. When data is supplied, names a column of it. (optional)
m_each	Use for a multi-arm design. A numeric vector giving the number of units (or clusters) assigned to each condition; must sum to N. (optional)
prob	Use for a two-arm design: either floor(N*prob) or ceiling(N*prob) units (or clusters) are assigned to treatment so that the marginal probability of assignment equals exactly prob. A single number between 0 and 1; use prob_unit to let it vary across units. (optional)

prob_unit
: Use for a two-arm design. Of length N. Under simple random assignment, may differ by unit or cluster. Under complete random assignment, must be constant across units. Under blocked random assignment, must be constant within blocks. Under balanced assignment (ra_type = "balanced"), may differ by unit. A single number is refused on every path, including the balanced one: use prob. When data is supplied, names a column of it. (optional)

prob_each
: Use for a multi-arm design. A numeric vector giving the probability of assignment to each condition; entries must be nonnegative and sum to 1. Due to integer rounding the exact count in each condition may differ slightly from draw to draw, but the overall probability is exactly prob_each. Under balanced assignment the same vector is expanded to one row per unit. (optional)

prob_unit_each
: Use for balanced assignment with two or more arms. A numeric matrix with one row per unit and one column per condition, giving each unit's probability of assignment to each condition. Rows must sum to 1. Supplying this argument selects balanced_ra(). When data is supplied, build it from columns, as in cbind(p_a, p_b). (optional)

block_m
: Use for a two-arm blocked design: a vector giving the number of units to assign to treatment within each block, in the order of sort(unique(blocks)). (optional)

block_m_each
: Use for a multi-arm blocked design. A matrix with one row per block and one column per treatment arm giving the number of units assigned to each condition within each block. Rows respect the ordering of sort(unique(blocks)). (optional)

block_prob
: Use for a two-arm blocked design in which the treatment probability varies across blocks. In the order of sort(unique(blocks)). (optional)

block_prob_each
: Use for a multi-arm blocked design in which treatment probabilities vary across blocks. A matrix with one row per block and one column per arm; each row must sum to 1. (optional)

num_arms
: The number of treatment arms. If unspecified, determined from the other arguments. (optional)

conditions
: A character vector giving the names of the treatment groups. If unspecified, groups will be named 0 and 1 in a two-arm trial and T1, T2, T3, in a multi-arm trial. A two-group design in which num_arms is set to 2 will use condition names T1 and T2. (optional)

simple
: Logical, defaults to FALSE. If TRUE, simple random assignment is used. Do not specify m, m_each, block_m, or block_m_each when simple = TRUE. (optional)

ra_type
: Optional override. The only accepted value is "balanced", which selects balanced_ra() and allows prob_unit to vary across units. Other designs are inferred from the arguments supplied; they cannot be forced with this argument. (optional)

formula
: For balanced assignment. A model formula whose model matrix is the balancing matrix X in the cube method, e.g. ~ x + B. The intercept is the count constraint. Do not also pass blocks. Supplying formula selects balanced_ra(). Two-arm only. The formula's variables are taken from data when it is supplied. They are looked up once, when the design is declared; conduct_ra() reuses the matrix

built then, so a later change to those variables does not change the declared design. (optional)

permutation_matrix
: For random assignment procedures that none of the other arguments can describe. A matrix with one row per unit and one column per assignment the procedure can produce, whose entries are condition names. Supplying it declares a design that draws one of those columns at random with equal probability, and the probabilities of assignment are read off the matrix by counting how often each unit appears in each condition. Build the matrix by calling your own assignment function many times and binding the results, or with `obtain_permutation_matrix()` for a design randomizr already knows. Ignored if `NULL`. (optional)

check_inputs
: Logical. Whether to verify before declaring that the arguments are internally consistent: that counts sum to N, that probabilities lie between 0 and 1 and sum to 1, that block-level arguments have one entry per block, and so on. Defaults to `TRUE`. `FALSE` skips the checking only: `num_arms` and `conditions` are still derived from the other arguments. It is skipped entirely when `permutation_matrix` is supplied. (optional)

data
: A data frame holding the design's variables. When supplied, every argument that carries one value per unit names columns of it and is looked up there and nowhere else: `blocks`, `clusters`, `m_unit`, `prob_unit`, `prob_unit_each`, and the variables in `formula`. Anything they name that is not a column is an error rather than a fall-through to the calling environment. A bare column name is the ordinary case; any expression works so long as every variable in it is a column, so `blocks = interaction(region, year)` and `prob_unit_each = cbind(p_a, p_b)` are fine and `blocks = df$bl` is not, because it names `df`. A string naming a column is also accepted. `N` defaults to `nrow(data)`. A declaration outlives the frame it was written in, so this is how to make it say exactly which variables it is built from. `permutation_matrix` is not resolved this way: it has one row per unit but enumerates assignments rather than describing units. When `data` is omitted, everything resolves in the calling environment as before. `data` itself is not stored in the declaration; the variables it supplies are. (optional)

Value

A vector of length N giving the treatment condition of each unit, numeric in a two-arm design and a factor (ordered by `conditions`) in a multi-arm design.

See Also

`declare_ra()`, `obtain_condition_probabilities()`

Examples

```
# Declare the design once, then draw from it
declaration <- declare_ra(N = 100, m_each = c(30, 30, 40))

Z <- conduct_ra(declaration = declaration)
```

```
table(Z)

# Equivalent, and convenient for a one-off assignment: describe the design
# inline and skip the declaration
Z <- conduct_ra(N = 100, m_each = c(30, 30, 40))
table(Z)
```

---

declare_ra *Declare a Random Assignment Procedure*

Description

declare_ra creates a reusable declaration object that captures all the parameters of a random assignment procedure. The declaration separates the specification of the design from the act of conducting it: call declare_ra once to fix the design, then call conduct_ra() repeatedly (for example, across simulation iterations) to draw assignments from the declared procedure. The declaration also precomputes and caches the probability of assignment for each unit, which obtain_condition_probabilities() returns for use in inverse-probability-weighted estimators.

Usage

```
declare_ra(
  N = NULL,
  blocks = NULL,
  clusters = NULL,
  m = NULL,
  m_unit = NULL,
  m_each = NULL,
  prob = NULL,
  prob_unit = NULL,
  prob_each = NULL,
  prob_unit_each = NULL,
  block_m = NULL,
  block_m_each = NULL,
  block_prob = NULL,
  block_prob_each = NULL,
  num_arms = NULL,
  conditions = NULL,
  simple = FALSE,
  ra_type = NULL,
  formula = NULL,
  permutation_matrix = NULL,
  check_inputs = TRUE,
  data = NULL
)
```

Arguments

N	The number of units. A positive integer. Optional when data, formula, or the length of prob_unit (or blocks, or clusters) identifies N.
blocks	A vector of length N indicating which block each unit belongs to, or, when data is supplied, the name of the column holding it. Supply to use blocked random assignment. (optional)
clusters	A vector of length N indicating which cluster each unit belongs to, or, when data is supplied, the name of the column holding it. Supply to use cluster random assignment. (optional)
m	Use for a two-arm design: exactly m units (or clusters) are assigned to treatment. In a blocked design, exactly m units in each block are treated. (optional)
m_unit	Use for a two-arm trial. A vector of length N; a single number is refused, since that is what m is for. Under complete random assignment, must be constant across units. Under blocked random assignment, must be constant within blocks. When data is supplied, names a column of it. (optional)
m_each	Use for a multi-arm design. A numeric vector giving the number of units (or clusters) assigned to each condition; must sum to N. (optional)
prob	Use for a two-arm design: either floor(N*prob) or ceiling(N*prob) units (or clusters) are assigned to treatment so that the marginal probability of assignment equals exactly prob. A single number between 0 and 1; use prob_unit to let it vary across units. (optional)
prob_unit	Use for a two-arm design. Of length N. Under simple random assignment, may differ by unit or cluster. Under complete random assignment, must be constant across units. Under blocked random assignment, must be constant within blocks. Under balanced assignment (ra_type = "balanced"), may differ by unit. A single number is refused on every path, including the balanced one: use prob. When data is supplied, names a column of it. (optional)
prob_each	Use for a multi-arm design. A numeric vector giving the probability of assignment to each condition; entries must be nonnegative and sum to 1. Due to integer rounding the exact count in each condition may differ slightly from draw to draw, but the overall probability is exactly prob_each. Under balanced assignment the same vector is expanded to one row per unit. (optional)
prob_unit_each	Use for balanced assignment with two or more arms. A numeric matrix with one row per unit and one column per condition, giving each unit's probability of assignment to each condition. Rows must sum to 1. Supplying this argument selects balanced_ra(). When data is supplied, build it from columns, as in cbind(p_a, p_b). (optional)
block_m	Use for a two-arm blocked design: a vector giving the number of units to assign to treatment within each block, in the order of sort(unique(blocks)). (optional)
block_m_each	Use for a multi-arm blocked design. A matrix with one row per block and one column per treatment arm giving the number of units assigned to each condition within each block. Rows respect the ordering of sort(unique(blocks)). (optional)

block_prob
: Use for a two-arm blocked design in which the treatment probability varies across blocks. In the order of `sort(unique(blocks))`. (optional)

block_prob_each
: Use for a multi-arm blocked design in which treatment probabilities vary across blocks. A matrix with one row per block and one column per arm; each row must sum to 1. (optional)

num_arms
: The number of treatment arms. If unspecified, determined from the other arguments. (optional)

conditions
: A character vector giving the names of the treatment groups. If unspecified, groups will be named 0 and 1 in a two-arm trial and T1, T2, T3, in a multi-arm trial. A two-group design in which `num_arms` is set to 2 will use condition names T1 and T2. (optional)

simple
: Logical, defaults to `FALSE`. If `TRUE`, simple random assignment is used. Do not specify `m`, `m_each`, `block_m`, or `block_m_each` when `simple = TRUE`. (optional)

ra_type
: Optional override. The only accepted value is `"balanced"`, which selects `balanced_ra()` and allows `prob_unit` to vary across units. Other designs are inferred from the arguments supplied; they cannot be forced with this argument. (optional)

formula
: For balanced assignment. A model formula whose model matrix is the balancing matrix X in the cube method, e.g. `~ x + B`. The intercept is the count constraint. Do not also pass `blocks`. Supplying `formula` selects `balanced_ra()`. Two-arm only. The formula's variables are taken from `data` when it is supplied. They are looked up once, when the design is declared; `conduct_ra()` reuses the matrix built then, so a later change to those variables does not change the declared design. (optional)

permutation_matrix
: For random assignment procedures that none of the other arguments can describe. A matrix with one row per unit and one column per assignment the procedure can produce, whose entries are condition names. Supplying it declares a design that draws one of those columns at random with equal probability, and the probabilities of assignment are read off the matrix by counting how often each unit appears in each condition. Build the matrix by calling your own assignment function many times and binding the results, or with `obtain_permutation_matrix()` for a design randomizr already knows. Ignored if `NULL`. (optional)

check_inputs
: Logical. Whether to verify before declaring that the arguments are internally consistent: that counts sum to N, that probabilities lie between 0 and 1 and sum to 1, that block-level arguments have one entry per block, and so on. Defaults to `TRUE`. `FALSE` skips the checking only: `num_arms` and `conditions` are still derived from the other arguments. It is skipped entirely when `permutation_matrix` is supplied. (optional)

data
: A data frame holding the design's variables. When supplied, every argument that carries one value per unit names columns of it and is looked up there and nowhere else: `blocks`, `clusters`, `m_unit`, `prob_unit`, `prob_unit_each`, and the variables in `formula`. Anything they name that is not a column is an error rather than a fall-through to the calling environment. A bare column name is the ordinary case; any expression works so long as every variable in it is

a column, so `blocks = interaction(region, year)` and `prob_unit_each = cbind(p_a, p_b)` are fine and `blocks = df$bl` is not, because it names `df`. A string naming a column is also accepted. `N` defaults to `nrow(data)`. A declaration outlives the frame it was written in, so this is how to make it say exactly which variables it is built from. `permutation_matrix` is not resolved this way: it has one row per unit but enumerates assignments rather than describing units. When `data` is omitted, everything resolves in the calling environment as before. `data` itself is not stored in the declaration; the variables it supplies are. (optional)

Details

`declare_ra` supports simple, complete, blocked, clustered, blocked-and-clustered, and balanced designs. It dispatches to the appropriate low-level function (`simple_ra()`, `complete_ra()`, `block_ra()`, `cluster_ra()`, `block_and_cluster_ra()`, or `balanced_ra()`) based on which arguments are supplied. Balanced assignment is opt-in: `declare_ra(N, prob = 0.5)` remains complete assignment. Use `ra_type = "balanced"` or supply `prob_unit_each` or `formula`.

Value

An object of class `"ra_declaration"` (an environment, addressable like a list) with entries:

`ra_function` A function that draws a random assignment from the declared procedure.

`ra_type` A string indicating the type of random assignment used.

`probabilities_matrix` A matrix with N rows and `num_arms` columns giving each unit's probability of assignment to each condition.

`blocks` The blocking variable, if supplied.

`clusters` The clustering variable, if supplied.

See Also

`conduct_ra()`, `obtain_condition_probabilities()`, `balanced_ra()`, `declare_rs()`

Examples

```
# A declaration is used in three ways.

# 1. To obtain some basic facts about a randomization:

declaration <- declare_ra(N = 100, m_each = c(30, 30, 40))
declaration

# 2. To conduct a random assignment:

Z <- conduct_ra(declaration)
table(Z)

# 3. To obtain the probability that each unit is in the condition it is in:

probs <- obtain_condition_probabilities(declaration, Z)
```

```
table(probs, Z)


# Simple Random Assignment Declarations

declare_ra(N = 100, simple = TRUE)

declare_ra(N = 100, prob = 0.4, simple = TRUE)

declare_ra(N = 100, prob_each = c(0.3, 0.3, 0.4),
           conditions = c("control", "placebo", "treatment"), simple = TRUE)


# Complete Random Assignment Declarations

declare_ra(N = 100)

declare_ra(N = 100, m_each = c(30, 70),
           conditions = c("control", "treatment"))

declare_ra(N = 100, m_each = c(30, 30, 40))


# Block Random Assignment Declarations

blocks <- rep(c("A", "B", "C"), times = c(50, 100, 200))
declare_ra(blocks = blocks)

# One row per block, one column per arm
block_m_each <- rbind(c(10, 40),
                      c(30, 70),
                      c(50, 150))

declare_ra(blocks = blocks, block_m_each = block_m_each)


# Cluster Random Assignment Declarations

clusters <- rep(letters[1:10], times = 1:10)

declare_ra(clusters = clusters)

declare_ra(clusters = clusters, m_each = c(3, 3, 4))


# Blocked and Clustered Random Assignment Declarations

clusters <- rep(letters[1:12], times = 1:12)

blocks <- rep(NA, length(clusters))
blocks[clusters %in% letters[1:3]] <- "block_1"
blocks[clusters %in% letters[4:6]] <- "block_2"
blocks[clusters %in% letters[7:9]] <- "block_3"
```

```
blocks[clusters %in% letters[10:12]] <- "block_4"

table(blocks, clusters)

declare_ra(clusters = clusters, blocks = blocks)

declare_ra(clusters = clusters, blocks = blocks, prob_each = c(0.2, 0.5, 0.3))


# Balanced assignment (tight counts; probabilities may vary).
# Opt-in: without ra_type or prob_unit_each this remains complete assignment.

p <- c(0.2, 0.4, 0.6, 0.8, 0.5, 0.5)
declare_ra(prob_unit = p, ra_type = "balanced")

P <- cbind(c(0.15, 0.47), c(0.65, 0.48), c(0.20, 0.05))
declare_ra(prob_unit_each = P)

x <- c(0, 1, 5, 6, 8, 9)
declare_ra(formula = ~ x)

# Name the table the design is built from, and blocks, clusters and the
# formula's variables are its columns rather than whatever the calling
# environment happens to hold.
dat <- data.frame(bl = rep(c("a", "b"), each = 3), x = c(0, 1, 5, 6, 8, 9),
                  p = c(0.2, 0.4, 0.5, 0.5, 0.6, 0.8))
declare_ra(blocks = bl, data = dat)
declare_ra(formula = ~ x, data = dat)
declare_ra(prob_unit = p, ra_type = "balanced", data = dat)
```

`declare_rs` *Declare a Random Sampling Procedure*

Description

`declare_rs` describes a sampling design once so that the rest of the package can work from it. Pass the result to `draw_rs()` to draw a sample, or to `obtain_inclusion_probabilities()` to recover each unit's probability of selection. Declaring is worth the extra line whenever a design is drawn more than once, since the probabilities are then computed from the same object that produced the sample rather than reconstructed by hand.

Usage

```
declare_rs(
  N = NULL,
  strata = NULL,
  clusters = NULL,
  n = NULL,
```

```
  n_unit = NULL,
  prob = NULL,
  prob_unit = NULL,
  strata_n = NULL,
  strata_prob = NULL,
  simple = FALSE,
  check_inputs = TRUE
)
```

Arguments

N	The number of units in the sampling frame. Must be a positive integer. (required)
strata	A vector of length N indicating which stratum each unit belongs to. Supply to use stratified random sampling. (optional)
clusters	A vector of length N indicating which cluster each unit belongs to. Supply to sample whole clusters. (optional)
n	Use for a design in which exactly n units (or clusters) are sampled. In a stratified design, exactly n units in each stratum are sampled. (optional)
n_unit	Of length N. Under complete random sampling, must be constant across units. Under stratified random sampling, must be constant within strata. (optional)
prob	Use for a design in which either floor(N*prob) or ceiling(N*prob) units (or clusters) are sampled. Which of the two is used is itself random: the ceiling is drawn with probability equal to the fractional part of N*prob and the floor otherwise, which makes each unit's probability of inclusion exactly prob. Must be a real number between 0 and 1 inclusive. (optional)
prob_unit	Of length N. Under simple random sampling, may differ for each unit or cluster. Under complete random sampling, must be constant across units. Under stratified random sampling, must be constant within strata. (optional)
strata_n	Use for a design in which strata_n gives the number of units to sample within each stratum, in the order of sort(unique(strata)). (optional)
strata_prob	Use for a design in which strata_prob gives the probability of being sampled within each stratum, in the order of sort(unique(strata)). Differs from prob in that the probability of being sampled can vary across strata. (optional)
simple	Logical, defaults to FALSE. If TRUE, simple random sampling is used, so the size of the realized sample varies from draw to draw. Do not specify n or strata_n when simple = TRUE; prob may then vary by unit. (optional)
check_inputs	Logical. Whether to verify before declaring that the arguments are internally consistent: that counts do not exceed the frame, that probabilities lie between 0 and 1, that stratum-level arguments have one entry per stratum, and so on. Defaults to TRUE. Set to FALSE to skip the checks when declaring many designs from arguments that have already been verified. (optional)

Details

declare_rs covers the same four designs as the sampling functions themselves: simple, complete, stratified, and clustered, in any combination. Which one it declares is inferred from the arguments given.

Value

An object of class "rs_declaration" (an environment, addressable like a list) with entries:

rs_function A function that draws a random sample from the declared procedure.

rs_type A string indicating the type of random sampling used.

probabilities_vector A vector of length N giving each unit's probability of being included in the sample.

strata The stratification variable, if supplied.

clusters The clustering variable, if supplied.

See Also

draw_rs(), obtain_inclusion_probabilities(), declare_ra()

Examples

```
# A declaration is used in three ways.

# 1. To obtain some basic facts about a sampling procedure:

declaration <- declare_rs(N = 100, n = 30)
declaration

# 2. To draw a random sample:

S <- draw_rs(declaration)
table(S)

# 3. To obtain inclusion probabilities:

probs <- obtain_inclusion_probabilities(declaration)
table(probs, S)


# Simple Random Sampling Declarations

declare_rs(N = 100, simple = TRUE)

declare_rs(N = 100, prob = 0.4, simple = TRUE)


# Complete Random Sampling Declarations

declare_rs(N = 100)
```

```
declare_rs(N = 100, n = 30)


# Stratified Random Sampling Declarations

strata <- rep(c("A", "B", "C"), times = c(50, 100, 200))

declare_rs(strata = strata)

declare_rs(strata = strata, prob = 0.5)


# Cluster Random Sampling Declarations

clusters <- rep(letters[1:10], times = 1:10)

declare_rs(clusters = clusters)

declare_rs(clusters = clusters, n = 4)


# Stratified and Clustered Random Sampling Declarations

clusters <- rep(letters[1:12], times = 1:12)

strata <- rep(NA, length(clusters))
strata[clusters %in% letters[1:3]] <- "stratum_1"
strata[clusters %in% letters[4:6]] <- "stratum_2"
strata[clusters %in% letters[7:9]] <- "stratum_3"
strata[clusters %in% letters[10:12]] <- "stratum_4"

table(strata, clusters)

declare_rs(clusters = clusters, strata = strata)

declare_rs(clusters = clusters, strata = strata, prob = 0.3)
```

`draw_rs` *Draw a Random Sample*

Description

`draw_rs` draws one random sample from a design. Give it a declaration made by `declare_rs()`, or describe the design inline with the same arguments `declare_rs()` takes. Declaring first pays off when the same design is drawn repeatedly, or when the inclusion probabilities are needed later by `obtain_inclusion_probabilities()`.

Usage

```
draw_rs(
  declaration = NULL,
  N = NULL,
  strata = NULL,
  clusters = NULL,
  n = NULL,
  n_unit = NULL,
  prob = NULL,
  prob_unit = NULL,
  strata_n = NULL,
  strata_prob = NULL,
  simple = FALSE,
  check_inputs = TRUE
)
```

Arguments

declaration	A random sampling declaration, created by declare_rs(). Supply either a declaration or the design arguments listed below, which are the ones declare_rs() takes: given those, draw_rs builds a declaration internally and draws one sample from it. (optional)
N	The number of units in the sampling frame. Must be a positive integer. (required)
strata	A vector of length N indicating which stratum each unit belongs to. Supply to use stratified random sampling. (optional)
clusters	A vector of length N indicating which cluster each unit belongs to. Supply to sample whole clusters. (optional)
n	Use for a design in which exactly n units (or clusters) are sampled. In a stratified design, exactly n units in each stratum are sampled. (optional)
n_unit	Of length N. Under complete random sampling, must be constant across units. Under stratified random sampling, must be constant within strata. (optional)
prob	Use for a design in which either floor(N*prob) or ceiling(N*prob) units (or clusters) are sampled. Which of the two is used is itself random: the ceiling is drawn with probability equal to the fractional part of N*prob and the floor otherwise, which makes each unit's probability of inclusion exactly prob. Must be a real number between 0 and 1 inclusive. (optional)
prob_unit	Of length N. Under simple random sampling, may differ for each unit or cluster. Under complete random sampling, must be constant across units. Under stratified random sampling, must be constant within strata. (optional)
strata_n	Use for a design in which strata_n gives the number of units to sample within each stratum, in the order of sort(unique(strata)). (optional)
strata_prob	Use for a design in which strata_prob gives the probability of being sampled within each stratum, in the order of sort(unique(strata)). Differs from prob in that the probability of being sampled can vary across strata. (optional)

simple	Logical, defaults to FALSE. If TRUE, simple random sampling is used, so the size of the realized sample varies from draw to draw. Do not specify n or strata_n when simple = TRUE; prob may then vary by unit. (optional)
check_inputs	Logical. Whether to verify before declaring that the arguments are internally consistent: that counts do not exceed the frame, that probabilities lie between 0 and 1, that stratum-level arguments have one entry per stratum, and so on. Defaults to TRUE. Set to FALSE to skip the checks when declaring many designs from arguments that have already been verified. (optional)

Value

A numeric vector of length N indicating whether each unit is sampled (1) or not (0).

See Also

declare_rs(), obtain_inclusion_probabilities()

Examples

```
# Declare the design once, then draw from it
declaration <- declare_rs(N = 100, n = 30)

S <- draw_rs(declaration = declaration)
table(S)

# Equivalent, and convenient for a one-off sample: describe the design
# inline and skip the declaration
S <- draw_rs(N = 100, n = 30)
table(S)
```

---

obtain_condition_probabilities

Obtain the Probability of the Condition Each Unit Is In

Description

A declaration holds the probability of every condition for every unit. obtain_condition_probabilities picks out, for each unit, the one probability that corresponds to the condition it was actually assigned to. Give it a declaration made by declare_ra(), or describe the design inline with the same arguments declare_ra() takes.

This function is especially useful when units have different probabilities of assignment and the analyst plans to use inverse-probability weights: the weights are the reciprocals of what it returns.

Usage

```
obtain_condition_probabilities(
  declaration = NULL,
  assignment,
  N = NULL,
  blocks = NULL,
  clusters = NULL,
  m = NULL,
  m_unit = NULL,
  m_each = NULL,
  prob = NULL,
  prob_unit = NULL,
  prob_each = NULL,
  prob_unit_each = NULL,
  block_m = NULL,
  block_m_each = NULL,
  block_prob = NULL,
  block_prob_each = NULL,
  num_arms = NULL,
  conditions = NULL,
  simple = FALSE,
  ra_type = NULL,
  formula = NULL,
  permutation_matrix = NULL,
  check_inputs = TRUE,
  data = NULL
)
```

Arguments

declaration	A random assignment declaration, created by declare_ra(). Supply either a declaration or the design arguments that declare_ra() takes. (optional)
assignment	A vector of random assignments, often created by conduct_ra(). (required)
N	The number of units. A positive integer. Optional when data, formula, or the length of prob_unit (or blocks, or clusters) identifies N.
blocks	A vector of length N indicating which block each unit belongs to, or, when data is supplied, the name of the column holding it. Supply to use blocked random assignment. (optional)
clusters	A vector of length N indicating which cluster each unit belongs to, or, when data is supplied, the name of the column holding it. Supply to use cluster random assignment. (optional)
m	Use for a two-arm design: exactly m units (or clusters) are assigned to treatment. In a blocked design, exactly m units in each block are treated. (optional)
m_unit	Use for a two-arm trial. A vector of length N; a single number is refused, since that is what m is for. Under complete random assignment, must be constant across units. Under blocked random assignment, must be constant within blocks. When data is supplied, names a column of it. (optional)

`m_each` Use for a multi-arm design. A numeric vector giving the number of units (or clusters) assigned to each condition; must sum to N. (optional)

`prob` Use for a two-arm design: either `floor(N*prob)` or `ceiling(N*prob)` units (or clusters) are assigned to treatment so that the marginal probability of assignment equals exactly `prob`. A single number between 0 and 1; use `prob_unit` to let it vary across units. (optional)

`prob_unit` Use for a two-arm design. Of length N. Under simple random assignment, may differ by unit or cluster. Under complete random assignment, must be constant across units. Under blocked random assignment, must be constant within blocks. Under balanced assignment (`ra_type = "balanced"`), may differ by unit. A single number is refused on every path, including the balanced one: use `prob`. When `data` is supplied, names a column of it. (optional)

`prob_each` Use for a multi-arm design. A numeric vector giving the probability of assignment to each condition; entries must be nonnegative and sum to 1. Due to integer rounding the exact count in each condition may differ slightly from draw to draw, but the overall probability is exactly `prob_each`. Under balanced assignment the same vector is expanded to one row per unit. (optional)

`prob_unit_each` Use for balanced assignment with two or more arms. A numeric matrix with one row per unit and one column per condition, giving each unit's probability of assignment to each condition. Rows must sum to 1. Supplying this argument selects `balanced_ra()`. When `data` is supplied, build it from columns, as in `cbind(p_a, p_b)`. (optional)

`block_m` Use for a two-arm blocked design: a vector giving the number of units to assign to treatment within each block, in the order of `sort(unique(blocks))`. (optional)

`block_m_each` Use for a multi-arm blocked design. A matrix with one row per block and one column per treatment arm giving the number of units assigned to each condition within each block. Rows respect the ordering of `sort(unique(blocks))`. (optional)

`block_prob` Use for a two-arm blocked design in which the treatment probability varies across blocks. In the order of `sort(unique(blocks))`. (optional)

`block_prob_each` Use for a multi-arm blocked design in which treatment probabilities vary across blocks. A matrix with one row per block and one column per arm; each row must sum to 1. (optional)

`num_arms` The number of treatment arms. If unspecified, determined from the other arguments. (optional)

`conditions` A character vector giving the names of the treatment groups. If unspecified, groups will be named 0 and 1 in a two-arm trial and T1, T2, T3, in a multi-arm trial. A two-group design in which `num_arms` is set to 2 will use condition names T1 and T2. (optional)

`simple` Logical, defaults to `FALSE`. If `TRUE`, simple random assignment is used. Do not specify `m`, `m_each`, `block_m`, or `block_m_each` when `simple = TRUE`. (optional)

`ra_type` Optional override. The only accepted value is `"balanced"`, which selects `balanced_ra()` and allows `prob_unit` to vary across units. Other designs are inferred from the arguments supplied; they cannot be forced with this argument. (optional)

`formula` For balanced assignment. A model formula whose model matrix is the balancing matrix X in the cube method, e.g. `~ x + B`. The intercept is the count constraint. Do not also pass `blocks`. Supplying `formula` selects `balanced_ra()`. Two-arm only. The formula's variables are taken from `data` when it is supplied. They are looked up once, when the design is declared; `conduct_ra()` reuses the matrix built then, so a later change to those variables does not change the declared design. (optional)

`permutation_matrix` For random assignment procedures that none of the other arguments can describe. A matrix with one row per unit and one column per assignment the procedure can produce, whose entries are condition names. Supplying it declares a design that draws one of those columns at random with equal probability, and the probabilities of assignment are read off the matrix by counting how often each unit appears in each condition. Build the matrix by calling your own assignment function many times and binding the results, or with `obtain_permutation_matrix()` for a design randomizr already knows. Ignored if `NULL`. (optional)

`check_inputs` Logical. Whether to verify before declaring that the arguments are internally consistent: that counts sum to N, that probabilities lie between 0 and 1 and sum to 1, that block-level arguments have one entry per block, and so on. Defaults to `TRUE`. `FALSE` skips the checking only: `num_arms` and `conditions` are still derived from the other arguments. It is skipped entirely when `permutation_matrix` is supplied. (optional)

`data` A data frame holding the design's variables. When supplied, every argument that carries one value per unit names columns of it and is looked up there and nowhere else: `blocks`, `clusters`, `m_unit`, `prob_unit`, `prob_unit_each`, and the variables in `formula`. Anything they name that is not a column is an error rather than a fall-through to the calling environment. A bare column name is the ordinary case; any expression works so long as every variable in it is a column, so `blocks = interaction(region, year)` and `prob_unit_each = cbind(p_a, p_b)` are fine and `blocks = df$bl` is not, because it names `df`. A string naming a column is also accepted. `N` defaults to `nrow(data)`. A declaration outlives the frame it was written in, so this is how to make it say exactly which variables it is built from. `permutation_matrix` is not resolved this way: it has one row per unit but enumerates assignments rather than describing units. When `data` is omitted, everything resolves in the calling environment as before. `data` itself is not stored in the declaration; the variables it supplies are. (optional)

Value

A vector of length N giving, for each unit, the probability that it was assigned to the condition it is actually in. These are the quantities inverse-probability weights are built from: weight each unit by the reciprocal of its value here.

See Also

`declare_ra()`, `conduct_ra()`

Examples

```
# Conduct a block random assignment in which the blocks have different
# probabilities of assignment to treatment
blocks <- rep(c("A", "B", "C"), times = c(50, 100, 200))

block_m_each <- rbind(c(10, 40),
                      c(30, 70),
                      c(50, 150))

declaration <- declare_ra(blocks = blocks, block_m_each = block_m_each)

Z <- conduct_ra(declaration = declaration)
table(Z, blocks)

observed_probabilities <-
   obtain_condition_probabilities(declaration = declaration, assignment = Z)

# Probabilities in the control group:
table(observed_probabilities[Z == 0], blocks[Z == 0])

# Probabilities in the treatment group:
table(observed_probabilities[Z == 1], blocks[Z == 1])

# The weights for an inverse-probability-weighted regression
ipw <- 1 / observed_probabilities


# Sometimes it is convenient to skip the declaration step
Z <- conduct_ra(blocks = blocks, block_m_each = block_m_each)

observed_probabilities <-
   obtain_condition_probabilities(assignment = Z,
                                  blocks = blocks,
                                  block_m_each = block_m_each)

table(observed_probabilities[Z == 0], blocks[Z == 0])
table(observed_probabilities[Z == 1], blocks[Z == 1])
```

---

obtain_inclusion_probabilities

Obtain Inclusion Probabilities

Description

Returns each unit's probability of being included in the sample under a declared design. Give `obtain_inclusion_probabilities()` a declaration made by `declare_rs()`, or describe the design inline with the same arguments `declare_rs()` takes.

This function is especially useful when units have different inclusion probabilities and the analyst plans to use inverse-probability weights: the weights are the reciprocals of what it returns.

Usage

```
obtain_inclusion_probabilities(
  declaration = NULL,
  N = NULL,
  strata = NULL,
  clusters = NULL,
  n = NULL,
  n_unit = NULL,
  prob = NULL,
  prob_unit = NULL,
  strata_n = NULL,
  strata_prob = NULL,
  simple = FALSE,
  check_inputs = TRUE
)
```

Arguments

declaration	A random sampling declaration, created by declare_rs(). Supply either a declaration or the design arguments that declare_rs() takes. (optional)
N	The number of units in the sampling frame. Must be a positive integer. (required)
strata	A vector of length N indicating which stratum each unit belongs to. Supply to use stratified random sampling. (optional)
clusters	A vector of length N indicating which cluster each unit belongs to. Supply to sample whole clusters. (optional)
n	Use for a design in which exactly n units (or clusters) are sampled. In a stratified design, exactly n units in each stratum are sampled. (optional)
n_unit	Of length N. Under complete random sampling, must be constant across units. Under stratified random sampling, must be constant within strata. (optional)
prob	Use for a design in which either floor(N*prob) or ceiling(N*prob) units (or clusters) are sampled. Which of the two is used is itself random: the ceiling is drawn with probability equal to the fractional part of N*prob and the floor otherwise, which makes each unit's probability of inclusion exactly prob. Must be a real number between 0 and 1 inclusive. (optional)
prob_unit	Of length N. Under simple random sampling, may differ for each unit or cluster. Under complete random sampling, must be constant across units. Under stratified random sampling, must be constant within strata. (optional)
strata_n	Use for a design in which strata_n gives the number of units to sample within each stratum, in the order of sort(unique(strata)). (optional)
strata_prob	Use for a design in which strata_prob gives the probability of being sampled within each stratum, in the order of sort(unique(strata)). Differs from prob in that the probability of being sampled can vary across strata. (optional)
simple	Logical, defaults to FALSE. If TRUE, simple random sampling is used, so the size of the realized sample varies from draw to draw. Do not specify n or strata_n when simple = TRUE; prob may then vary by unit. (optional)

check_inputs Logical. Whether to verify before declaring that the arguments are internally consistent: that counts do not exceed the frame, that probabilities lie between 0 and 1, that stratum-level arguments have one entry per stratum, and so on. Defaults to TRUE. Set to FALSE to skip the checks when declaring many designs from arguments that have already been verified. (optional)

Value

A numeric vector of length N giving each unit's probability of being included in the sample. These are the quantities inverse-probability weights are built from: weight each sampled unit by the reciprocal of its value here.

See Also

declare_rs(), draw_rs()

Examples

```
# A stratified design in which the strata are sampled at different rates
strata <- rep(c("A", "B", "C"), times = c(50, 100, 200))

declaration <- declare_rs(strata = strata, strata_n = c(20, 30, 40))

observed_probabilities <-
   obtain_inclusion_probabilities(declaration = declaration)

table(strata, observed_probabilities)

# The weights for an inverse-probability-weighted analysis
ipw <- 1 / observed_probabilities


# Sometimes it is convenient to skip the declaration step
observed_probabilities <-
   obtain_inclusion_probabilities(strata = strata, strata_n = c(20, 30, 40))

table(strata, observed_probabilities)
```

---

obtain_num_permutations

Obtain the Number of Possible Permutations from a Random Assignment Declaration

---

Description

Counts the assignments a design could have produced. The count is the size of the randomization distribution, so it says how much resolution a randomization inference p-value can have: a design with 70 possible assignments cannot produce a p-value below 1/70. Counting is exact and cheap even when the number is far too large to enumerate, which is why it is worth calling before obtain_permutation_matrix().

Usage

```
obtain_num_permutations(declaration)
```

Arguments

declaration	A random assignment or sampling declaration, created by declare_ra() or declare_rs(). (required)

Value

A single number: how many distinct assignments (or samples) the declared design can produce. It can be far larger than any matrix you would want to build, which is the point of counting first.

See Also

obtain_permutation_matrix(), obtain_permutation_probabilities(), declare_ra()

Examples

```
# Random assignment
## complete

declaration <- declare_ra(N = 4)
perms <- obtain_permutation_matrix(declaration)
dim(perms)
obtain_num_permutations(declaration)

## blocked

blocks <- c("A", "A", "B", "B", "C", "C", "C")
declaration <- declare_ra(blocks = blocks)
perms <- obtain_permutation_matrix(declaration)
dim(perms)
obtain_num_permutations(declaration)

## clustered

clusters <- c("A", "B", "A", "B", "C", "C", "C")
declaration <- declare_ra(clusters = clusters)
perms <- obtain_permutation_matrix(declaration)
dim(perms)
obtain_num_permutations(declaration)

## large

declaration <- declare_ra(20)
choose(20, 10)
perms <- obtain_permutation_matrix(declaration)
dim(perms)

# Random sampling
## complete
```

```
declaration <- declare_rs(N = 4)
perms <- obtain_permutation_matrix(declaration)
dim(perms)
obtain_num_permutations(declaration)

## stratified

strata <- c("A", "A", "B", "B", "C", "C", "C")
declaration <- declare_rs(strata = strata)
perms <- obtain_permutation_matrix(declaration)
dim(perms)
obtain_num_permutations(declaration)

## clustered

clusters <- c("A", "B", "A", "B", "C", "C", "C")
declaration <- declare_rs(clusters = clusters)
perms <- obtain_permutation_matrix(declaration)
dim(perms)
obtain_num_permutations(declaration)

## large

declaration <- declare_rs(N = 20)
perms <- obtain_permutation_matrix(declaration)
dim(perms)
```

`obtain_permutation_matrix` *Obtain Permutation Matrix from a Random Assignment Declaration*

Description

Enumerates the assignments a design could have produced, one column per assignment. The matrix is the input to randomization inference, where a test statistic is recomputed under each column to build the distribution it would follow if the treatment had no effect. When a design admits more assignments than `maximum_permutations`, a random sample of them is returned instead, which approximates the same distribution.

Usage

```
obtain_permutation_matrix(declaration, maximum_permutations = 10000)
```

Arguments

`declaration`	A random assignment declaration, created by `declare_ra()`. (required)

maximum_permutations
: If the number of possible random assignments exceeds maximum_permutations, obtain_permutation_matrix returns a random sample of maximum_permutations of them instead of enumerating all of them. Defaults to 10,000. (optional)

Value

A matrix with one row per unit and one column per assignment, whose entries are condition names. The columns are all of the assignments the declared design could produce, or a random sample of maximum_permutations of them if there are more than that. Column order carries no meaning, but it is the order obtain_permutation_probabilities() returns its probabilities in.

References

Andrews, G. E. (1976). *The Theory of Partitions*. Encyclopedia of Mathematics and its Applications, Volume 2. Reading, MA: Addison-Wesley.

See Also

obtain_num_permutations(), obtain_permutation_probabilities(), declare_ra()

Examples

```
# complete

declaration <- declare_ra(N = 4)
perms <- obtain_permutation_matrix(declaration)
dim(perms)
obtain_num_permutations(declaration)

# blocked

blocks <- c("A", "A", "B", "B", "C", "C", "C")
declaration <- declare_ra(blocks = blocks)
perms <- obtain_permutation_matrix(declaration)
dim(perms)
obtain_num_permutations(declaration)

# clustered

clusters <- c("A", "B", "A", "B", "C", "C", "C")
declaration <- declare_ra(clusters = clusters)
perms <- obtain_permutation_matrix(declaration)
dim(perms)
obtain_num_permutations(declaration)

# large

declaration <- declare_ra(20)
choose(20, 10)
perms <- obtain_permutation_matrix(declaration)
dim(perms)
```

`obtain_permutation_probabilities` *Obtain the probabilities of permutations*

Description

Returns how likely each assignment in the permutation matrix was. Most designs make every possible assignment equally likely, in which case these are all the same and can be ignored. Blocked and clustered designs of unequal size do not, and there the probabilities are needed to weight the randomization distribution correctly.

Usage

```
obtain_permutation_probabilities(declaration)
```

Arguments

`declaration`	A random assignment declaration, created by `declare_ra()`. (required)

Value

A vector with one entry per possible assignment, giving the probability that the design produces that assignment. The entries sum to 1 and are in the same order as the columns of `obtain_permutation_matrix()`, so the two can be used together.

References

Andrews, G. E. (1976). *The Theory of Partitions*. Encyclopedia of Mathematics and its Applications, Volume 2. Reading, MA: Addison-Wesley.

See Also

`obtain_permutation_matrix()`, `obtain_num_permutations()`

Examples

```
# A design in which the possible assignments are *not* equally likely: with
# N = 5 and prob = 0.51, either 2 or 3 units are treated, and those two cases
# do not come up equally often.
declaration <- declare_ra(N = 5, prob_each = c(0.49, 0.51))

obtain_num_permutations(declaration)

perms <- obtain_permutation_matrix(declaration)
perm_probs <- obtain_permutation_probabilities(declaration)
```

```
# perms has one column per possible assignment and perm_probs has one entry
# per column, in the same order
dim(perms)
length(perm_probs)

# Each unit's probability of assignment to treatment, according to the
# declaration. Recovering these from perms is the check that the two objects
# line up.
true_probabilities <- declaration$probabilities_matrix[, 2]
true_probabilities

# The unweighted average across columns is WRONG here: it treats every
# assignment as equally likely, which this design does not.
rowMeans(perms)

# Weighting each column by how likely it is recovers the true probabilities.
# The matrix product does the weighted average: row i of perms times
# perm_probs sums unit i's treatment indicators weighted by column
# probability, which is exactly Pr(unit i treated).
perms %*% perm_probs
```

`simple_ra` *Simple Random Assignment*

Description

`simple_ra` assigns units to treatment conditions independently, with each unit's assignment drawn as a separate Bernoulli trial. Because units are assigned independently, the number of units assigned to each condition varies from draw to draw. For most experimental applications in which the number of units is known in advance, `complete_ra()` is preferable because it fixes the counts in each condition and thereby reduces sampling variability.

Usage

```
simple_ra(
  N,
  prob = NULL,
  prob_unit = NULL,
  prob_each = NULL,
  num_arms = NULL,
  conditions = NULL,
  check_inputs = TRUE,
  simple = TRUE
)
```

Arguments

N	The number of units. Must be a positive integer. (required)
prob	Use for a two-arm design. The probability of assignment to treatment; must be a real number between 0 and 1 and of length 1. (optional)
prob_unit	Use for a two-arm design. The probability of assignment to treatment for each unit; must be a real number between 0 and 1 and of length N. (optional)
prob_each	Use for a multi-arm design. A numeric vector or N-by-conditions matrix giving the probability of assignment to each condition; entries must be nonnegative and sum to 1. (optional)
num_arms	The number of treatment arms. If unspecified, determined from the other arguments. (optional)
conditions	A character vector giving the names of the treatment groups. If unspecified, groups will be named 0 and 1 in a two-arm trial and T1, T2, T3, in a multi-arm trial. A two-group design in which num_arms is set to 2 will use condition names T1 and T2. (optional)
check_inputs	Logical. Whether to verify before assigning that the arguments are internally consistent: that probabilities lie between 0 and 1 and sum to 1, that vectors are of length N, that only one of prob, prob_unit, and prob_each is supplied, and so on. Defaults to TRUE. FALSE skips the checking only: num_arms and conditions are still derived from the other arguments, so the same call draws the same assignment either way. What goes is the verification, and an impossible design is then no longer refused. block_m larger than a block, for instance, quietly treats the whole block. Declaring the design once with declare_ra() and drawing from it with conduct_ra() is the usual way to avoid re-checking the same arguments in a simulation. (optional)
simple	Logical. Internal use only; leave at its default. simple_ra always assigns units independently, and this argument exists so that the argument checker knows as much. Setting it to FALSE does not change how units are assigned, but it will cause a prob_unit that varies across units to be rejected. (optional)

Details

Simple random assignment is appropriate when units arrive sequentially and the total sample size is not known in advance, or when the assignment must proceed without coordinating across units. If only N is specified, a two-arm trial with prob = 0.5 is assumed.

Value

A vector of length N indicating the treatment condition of each unit. Numeric in a two-arm trial; a factor (ordered by conditions) in a multi-arm trial.

See Also

complete_ra(), block_ra(), simple_rs(), simple_ra_probabilities()

Examples

```
# Two Group Designs

Z <- simple_ra(N = 100)
table(Z)

Z <- simple_ra(N = 100, prob = 0.5)
table(Z)

Z <- simple_ra(N = 100, prob_each = c(0.3, 0.7),
               conditions = c("control", "treatment"))
table(Z)

# A probability of assignment that varies unit by unit
Z <- simple_ra(N = 100, prob_unit = seq(0.1, 0.9, length.out = 100))
table(Z)

# Skipping the input checks. The checks are also what fill in defaults, so
# conditions has to be given explicitly once they are skipped. In a
# simulation, declare_ra() and conduct_ra() are the tidier way to check the
# arguments once and then draw many assignments from them.
Z <- simple_ra(N = 100, prob = 0.3, conditions = c(0, 1), check_inputs = FALSE)
table(Z)

# Multi-arm Designs
Z <- simple_ra(N = 100, num_arms = 3)
table(Z)

Z <- simple_ra(N = 100, prob_each = c(0.3, 0.3, 0.4))
table(Z)

Z <- simple_ra(N = 100, prob_each = c(0.3, 0.3, 0.4),
               conditions = c("control", "placebo", "treatment"))
table(Z)

Z <- simple_ra(N = 100, conditions = c("control", "placebo", "treatment"))
table(Z)
```

---

simple_ra_probabilities

Probabilities of assignment: Simple Random Assignment

Description

Returns the probability that each unit is assigned to each condition under simple random assignment. Every unit is assigned independently, so the probabilities do not depend on how the other units came out.

Usage

```
simple_ra_probabilities(
  N,
  prob = NULL,
  prob_unit = NULL,
  prob_each = NULL,
  num_arms = NULL,
  conditions = NULL,
  check_inputs = TRUE,
  simple = TRUE
)
```

Arguments

N	The number of units. Must be a positive integer. (required)
prob	Use for a two-arm design. The probability of assignment to treatment; must be a real number between 0 and 1 and of length 1. (optional)
prob_unit	Use for a two-arm design. The probability of assignment to treatment for each unit; must be a real number between 0 and 1 and of length N. (optional)
prob_each	Use for a multi-arm design. A numeric vector or N-by-conditions matrix giving the probability of assignment to each condition; entries must be nonnegative and sum to 1. (optional)
num_arms	The number of treatment arms. If unspecified, determined from the other arguments. (optional)
conditions	A character vector giving the names of the treatment groups. If unspecified, groups will be named 0 and 1 in a two-arm trial and T1, T2, T3, in a multi-arm trial. A two-group design in which num_arms is set to 2 will use condition names T1 and T2. (optional)
check_inputs	Logical. Whether to verify before assigning that the arguments are internally consistent: that probabilities lie between 0 and 1 and sum to 1, that vectors are of length N, that only one of prob, prob_unit, and prob_each is supplied, and so on. Defaults to TRUE. FALSE skips the checking only: num_arms and conditions are still derived from the other arguments, so the same call draws the same assignment either way. What goes is the verification, and an impossible design is then no longer refused. block_m larger than a block, for instance, quietly treats the whole block. Declaring the design once with declare_ra() and drawing from it with conduct_ra() is the usual way to avoid re-checking the same arguments in a simulation. (optional)
simple	Logical. Internal use only; leave at its default. simple_ra always assigns units independently, and this argument exists so that the argument checker knows as much. Setting it to FALSE does not change how units are assigned, but it will cause a prob_unit that varies across units to be rejected. (optional)

Details

These are the quantities inverse-probability weights are built from: weight each unit by the reciprocal of the probability of the condition it landed in, which obtain_condition_probabilities()

extracts for you.

Value

A matrix with N rows and one column per treatment condition, with columns named `prob_<condition>`. Entry (i, j) is the probability that unit i is assigned to condition j, and every row sums to 1.

See Also

`simple_ra()`

Examples

```
# Two Group Designs
prob_mat <- simple_ra_probabilities(N = 100)
head(prob_mat)

prob_mat <- simple_ra_probabilities(N = 100, prob = 0.5)
head(prob_mat)

prob_mat <- simple_ra_probabilities(N = 100, prob_each = c(0.3, 0.7),
                        conditions = c("control", "treatment"))
head(prob_mat)

# Multi-arm Designs
prob_mat <- simple_ra_probabilities(N = 100, num_arms = 3)
head(prob_mat)

prob_mat <- simple_ra_probabilities(N = 100, prob_each = c(0.3, 0.3, 0.4))
head(prob_mat)

prob_mat <- simple_ra_probabilities(N = 100, prob_each = c(0.3, 0.3, 0.4),
                        conditions = c("control", "placebo", "treatment"))
head(prob_mat)

prob_mat <- simple_ra_probabilities(N = 100, conditions = c("control", "placebo", "treatment"))
head(prob_mat)
```

---

simple_rs *Simple Random Sampling*

Description

`simple_rs` draws a sample in which every unit is included or not independently of the others, as a separate coin flip. Because the draws are independent, the size of the realized sample varies from draw to draw. For most applications in which the size of the sampling frame is known in advance, `complete_rs()` is preferable because it fixes the number of units sampled.

Usage

```
simple_rs(N, prob = NULL, prob_unit = NULL, check_inputs = TRUE, simple = TRUE)
```

Arguments

N	The number of units in the sampling frame. Must be a positive integer. (required)
prob	The probability of being sampled; must be a real number between 0 and 1 inclusive and of length 1. (optional)
prob_unit	The probability of being sampled for each unit; must be a real number between 0 and 1 inclusive and of length N. Because units are drawn independently, this probability may differ from unit to unit. (optional)
check_inputs	Logical. Whether to verify before sampling that the arguments are internally consistent: that probabilities lie between 0 and 1, that vectors are of length N, and that only one of prob and prob_unit is supplied. Defaults to TRUE. Set to FALSE to skip the checks when drawing many samples from arguments that have already been verified; declaring the design once with declare_rs() and drawing from it with draw_rs() does this for you. (optional)
simple	Logical. Internal use only; leave at its default. simple_rs always draws units independently, and this argument exists so that the argument checker knows as much. (optional)

Details

If prob is not specified, each unit is sampled with probability 0.5.

Value

A numeric vector of length N indicating whether each unit is sampled (1) or not (0).

See Also

complete_rs(), strata_rs(), simple_ra(), simple_rs_probabilities()

Examples

```
S <- simple_rs(N = 100)
table(S)

S <- simple_rs(N = 100, prob = 0.3)
table(S)

# A probability of inclusion that varies unit by unit
S <- simple_rs(N = 100, prob_unit = seq(0.1, 0.9, length.out = 100))
table(S)
```

`simple_rs_probabilities` *Inclusion probabilities: Simple Random Sampling*

Description

Returns each unit's probability of being sampled under simple random sampling. Every unit is sampled independently, so the probabilities do not depend on which other units were drawn.

Usage

```
simple_rs_probabilities(
  N,
  prob = NULL,
  prob_unit = NULL,
  check_inputs = TRUE,
  simple = TRUE
)
```

Arguments

`N`	The number of units in the sampling frame. Must be a positive integer. (required)
`prob`	The probability of being sampled; must be a real number between 0 and 1 inclusive and of length 1. (optional)
`prob_unit`	The probability of being sampled for each unit; must be a real number between 0 and 1 inclusive and of length N. Because units are drawn independently, this probability may differ from unit to unit. (optional)
`check_inputs`	Logical. Whether to verify before sampling that the arguments are internally consistent: that probabilities lie between 0 and 1, that vectors are of length N, and that only one of `prob` and `prob_unit` is supplied. Defaults to `TRUE`. Set to `FALSE` to skip the checks when drawing many samples from arguments that have already been verified; declaring the design once with `declare_rs()` and drawing from it with `draw_rs()` does this for you. (optional)
`simple`	Logical. Internal use only; leave at its default. `simple_rs` always draws units independently, and this argument exists so that the argument checker knows as much. (optional)

Details

These are the quantities inverse-probability weights are built from: weight each sampled unit by the reciprocal of its inclusion probability, which `obtain_inclusion_probabilities()` extracts for you.

Value

A numeric vector of length N giving each unit's probability of being included in the sample.

See Also

simple_rs()

Examples

```
probs <- simple_rs_probabilities(N = 100)
table(probs)

probs <- simple_rs_probabilities(N = 100, prob = 0.3)
table(probs)
```

---

strata_and_cluster_rs *Stratified and Clustered Random Sampling*

Description

strata_and_cluster_rs draws whole clusters, sampling separately within each stratum. Use it when the sampling unit is a group rather than an individual and the groups fall into categories you want represented in fixed proportion. Sampling by cluster costs precision, since the effective sample size is the number of clusters rather than the number of units; stratifying buys some of it back by fixing how many clusters come from each stratum.

Usage

```
strata_and_cluster_rs(
  strata = NULL,
  clusters = NULL,
  prob = NULL,
  prob_unit = NULL,
  n = NULL,
  n_unit = NULL,
  strata_n = NULL,
  strata_prob = NULL,
  check_inputs = TRUE
)
```

Arguments

strata	A vector of length N indicating which stratum each unit belongs to. Every unit in a cluster must belong to the same stratum. (required)
clusters	A vector of length N indicating which cluster each unit belongs to. (required)
prob	Use for a design in which either floor(N_clusters_stratum*prob) or ceiling(N_clusters_stratum clusters are sampled within each stratum. Which of the two is used is itself random: the ceiling is drawn with probability equal to the fractional part of N_clusters_stratum*prob and the floor otherwise, which makes each cluster's probability of inclusion exactly prob. Must be a real number between 0 and 1 inclusive. (optional)

prob_unit	Must be of length N. `tapply(prob_unit, strata, unique)` will be passed to `strata_prob`, so it must be constant within each stratum. (optional)
n	Use for a design in which the scalar `n` gives the fixed number of clusters to sample in every stratum. This count does not vary across strata. (optional)
n_unit	Must be of length N. `tapply(n_unit, strata, unique)` will be passed to `strata_n`, so it must be constant within each stratum. (optional)
strata_n	Use for a design in which `strata_n` gives the number of clusters to sample within each stratum. Must be as long as the number of strata, in the same order as `sort(unique(strata))`. (optional)
strata_prob	Use for a design in which `strata_prob` gives the probability of being sampled within each stratum. Must be in the same order as `sort(unique(strata))`. Differs from `prob` in that the probability of being sampled can vary across strata. (optional)
check_inputs	Logical. Whether to verify before sampling that the arguments are internally consistent: that clusters nest within strata, that counts do not exceed the number of clusters in a stratum, that probabilities lie between 0 and 1, and so on. Defaults to `TRUE`. Set to `FALSE` to skip the checks when drawing many samples from arguments that have already been verified; declaring the design once with `declare_rs()` and drawing from it with `draw_rs()` does this for you. (optional)

Details

Clusters must nest within strata: every unit in a cluster has to belong to the same stratum.

Value

A numeric vector of length N indicating whether each unit is sampled (1) or not (0). Every unit in a cluster receives the same value.

See Also

`cluster_rs()`, `strata_rs()`, `block_and_cluster_ra()`

Examples

```
# Twelve clusters, of sizes 1 through 12, nested in four strata of three
clusters <- rep(letters[1:12], times = 1:12)

strata <- rep(NA, length(clusters))
strata[clusters %in% letters[1:3]] <- "stratum_1"
strata[clusters %in% letters[4:6]] <- "stratum_2"
strata[clusters %in% letters[7:9]] <- "stratum_3"
strata[clusters %in% letters[10:12]] <- "stratum_4"

table(strata, clusters)

S <- strata_and_cluster_rs(strata = strata,
                           clusters = clusters)
```

```
table(S, strata)
table(S, clusters)


S <- strata_and_cluster_rs(clusters = clusters,
                           strata = strata,
                           prob = 0.5)

table(S, clusters)
table(S, strata)

S <- strata_and_cluster_rs(clusters = clusters,
                           strata = strata,
                           strata_n = c(1, 2, 1, 2))

table(S, clusters)
table(S, strata)

S <- strata_and_cluster_rs(clusters = clusters,
                           strata = strata,
                           strata_prob = c(0.2, 0.4, 0.6, 0.8))

table(S, clusters)
table(S, strata)
```

`strata_and_cluster_rs_probabilities` *Inclusion probabilities: Stratified and Clustered Random Sampling*

Description

Returns each unit's probability of being sampled when clusters are drawn within strata. Probabilities vary across strata and are constant within a cluster.

Usage

```
strata_and_cluster_rs_probabilities(
  strata = NULL,
  clusters = NULL,
  prob = NULL,
  prob_unit = NULL,
  n = NULL,
  n_unit = NULL,
  strata_n = NULL,
  strata_prob = NULL,
  check_inputs = TRUE
)
```

Arguments

strata	A vector of length N indicating which stratum each unit belongs to. Every unit in a cluster must belong to the same stratum. (required)
clusters	A vector of length N indicating which cluster each unit belongs to. (required)
prob	Use for a design in which either floor(N_clusters_stratum*prob) or ceiling(N_clusters_stratum clusters are sampled within each stratum. Which of the two is used is itself random: the ceiling is drawn with probability equal to the fractional part of N_clusters_stratum*prob and the floor otherwise, which makes each cluster's probability of inclusion exactly prob. Must be a real number between 0 and 1 inclusive. (optional)
prob_unit	Must be of length N. tapply(prob_unit, strata, unique) will be passed to strata_prob, so it must be constant within each stratum. (optional)
n	Use for a design in which the scalar n gives the fixed number of clusters to sample in every stratum. This count does not vary across strata. (optional)
n_unit	Must be of length N. tapply(n_unit, strata, unique) will be passed to strata_n, so it must be constant within each stratum. (optional)
strata_n	Use for a design in which strata_n gives the number of clusters to sample within each stratum. Must be as long as the number of strata, in the same order as sort(unique(strata)). (optional)
strata_prob	Use for a design in which strata_prob gives the probability of being sampled within each stratum. Must be in the same order as sort(unique(strata)). Differs from prob in that the probability of being sampled can vary across strata. (optional)
check_inputs	Logical. Whether to verify before sampling that the arguments are internally consistent: that clusters nest within strata, that counts do not exceed the number of clusters in a stratum, that probabilities lie between 0 and 1, and so on. Defaults to TRUE. Set to FALSE to skip the checks when drawing many samples from arguments that have already been verified; declaring the design once with declare_rs() and drawing from it with draw_rs() does this for you. (optional)

Details

These are the quantities inverse-probability weights are built from: weight each sampled unit by the reciprocal of its inclusion probability, which obtain_inclusion_probabilities() extracts for you.

Value

A numeric vector of length N giving each unit's probability of being included in the sample. Every unit in a cluster shares one probability.

See Also

strata_and_cluster_rs()

Examples

```
# Twelve clusters, of sizes 1 through 12, nested in four strata of three
clusters <- rep(letters[1:12], times = 1:12)

strata <- rep(NA, length(clusters))
strata[clusters %in% letters[1:3]] <- "stratum_1"
strata[clusters %in% letters[4:6]] <- "stratum_2"
strata[clusters %in% letters[7:9]] <- "stratum_3"
strata[clusters %in% letters[10:12]] <- "stratum_4"

table(strata, clusters)

probs <- strata_and_cluster_rs_probabilities(strata = strata,
                                             clusters = clusters)

table(probs, strata)
table(probs, clusters)


probs <- strata_and_cluster_rs_probabilities(clusters = clusters,
                                             strata = strata,
                                             prob = 0.5)

table(probs, clusters)
table(probs, strata)

probs <- strata_and_cluster_rs_probabilities(clusters = clusters,
                                             strata = strata,
                                             strata_n = c(1, 2, 1, 2))

table(probs, clusters)
table(probs, strata)

probs <- strata_and_cluster_rs_probabilities(clusters = clusters,
                                             strata = strata,
                                             strata_prob = c(0.2, 0.4, 0.6, 0.8))

table(probs, clusters)
table(probs, strata)
```

`strata_rs` *Stratified Random Sampling*

Description

`strata_rs` draws a sample separately within each of several groups (strata) defined by covariates, using complete random sampling inside every stratum. For example, 50 of 100 men and 75 of 200 women might be sampled. Stratifying guarantees how much of the sample comes from each group, which keeps small groups from being underrepresented by chance.

Usage

```
strata_rs(
  strata = NULL,
  prob = NULL,
  prob_unit = NULL,
  n = NULL,
  n_unit = NULL,
  strata_n = NULL,
  strata_prob = NULL,
  check_inputs = TRUE
)
```

Arguments

strata	A vector of length N indicating which stratum each unit belongs to. Can be a character, factor, or numeric vector. (required)
prob	Use for a design in which either `floor(N_stratum*prob)` or `ceiling(N_stratum*prob)` units are sampled within each stratum. Which of the two is used is itself random: the ceiling is drawn with probability equal to the fractional part of `N_stratum*prob` and the floor otherwise, which makes each unit's probability of inclusion exactly `prob`. Must be a real number between 0 and 1 inclusive. (optional)
prob_unit	Must be of length N. `tapply(prob_unit, strata, unique)` will be passed to `strata_prob`, so it must be constant within each stratum. (optional)
n	Use for a design in which the scalar `n` gives the fixed number of units to sample in every stratum. This count does not vary across strata. (optional)
n_unit	Must be of length N. `tapply(n_unit, strata, unique)` will be passed to `strata_n`, so it must be constant within each stratum. (optional)
strata_n	Use for a design in which the numeric vector `strata_n` gives the number of units to sample within each stratum. Must be as long as the number of strata, in the same order as `sort(unique(strata))`. (optional)
strata_prob	Use for a design in which `strata_prob` gives the probability of being sampled within each stratum. Must be in the same order as `sort(unique(strata))`. Differs from `prob` in that the probability of being sampled can vary across strata. (optional)
check_inputs	Logical. Whether to verify before sampling that the arguments are internally consistent: that counts do not exceed the stratum sizes, that probabilities lie between 0 and 1, that stratum-level arguments have one entry per stratum, and so on. Defaults to `TRUE`. Set to `FALSE` to skip the checks when drawing many samples from arguments that have already been verified; declaring the design once with `declare_rs()` and drawing from it with `draw_rs()` does this for you. (optional)

Details

The number sampled per stratum can be left to the function, set as a common count or probability across strata (`n`, `prob`), or set stratum by stratum (`strata_n`, `strata_prob`). When the probability

varies across strata the sample is not self-weighting, and strata_rs_probabilities() gives the inclusion probabilities needed to weight it.

Value

A numeric vector of length N indicating whether each unit is sampled (1) or not (0).

See Also

complete_rs(), strata_and_cluster_rs(), block_ra(), strata_rs_probabilities()

Examples

```
strata <- rep(c("A", "B", "C"), times = c(50, 100, 200))

S <- strata_rs(strata = strata)
table(strata, S)

# The same probability in every stratum
S <- strata_rs(strata = strata, prob = 0.3)
table(strata, S)

# The same count in every stratum
S <- strata_rs(strata = strata, n = 20)
table(strata, S)

# A different probability in each stratum, in the order of sort(unique(strata))
S <- strata_rs(strata = strata, strata_prob = c(0.1, 0.2, 0.3))
table(strata, S)

# The same, specified unit by unit
S <- strata_rs(strata = strata,
               prob_unit = rep(c(0.1, 0.2, 0.3), times = c(50, 100, 200)))
table(strata, S)

# A different count in each stratum
S <- strata_rs(strata = strata, strata_n = c(20, 30, 40))
table(strata, S)

S <- strata_rs(strata = strata,
               n_unit = rep(c(20, 30, 40), times = c(50, 100, 200)))
table(strata, S)
```

---

strata_rs_probabilities

Inclusion probabilities: Stratified Random Sampling

Description

Returns each unit's probability of being sampled under stratified random sampling. Units in different strata routinely have different probabilities, and a sample drawn that way is not self-weighting.

Usage

```
strata_rs_probabilities(
  strata = NULL,
  prob = NULL,
  prob_unit = NULL,
  n = NULL,
  n_unit = NULL,
  strata_n = NULL,
  strata_prob = NULL,
  check_inputs = TRUE
)
```

Arguments

`strata`	A vector of length N indicating which stratum each unit belongs to. Can be a character, factor, or numeric vector. (required)
`prob`	Use for a design in which either `floor(N_stratum*prob)` or `ceiling(N_stratum*prob)` units are sampled within each stratum. Which of the two is used is itself random: the ceiling is drawn with probability equal to the fractional part of `N_stratum*prob` and the floor otherwise, which makes each unit's probability of inclusion exactly `prob`. Must be a real number between 0 and 1 inclusive. (optional)
`prob_unit`	Must be of length N. `tapply(prob_unit, strata, unique)` will be passed to `strata_prob`, so it must be constant within each stratum. (optional)
`n`	Use for a design in which the scalar `n` gives the fixed number of units to sample in every stratum. This count does not vary across strata. (optional)
`n_unit`	Must be of length N. `tapply(n_unit, strata, unique)` will be passed to `strata_n`, so it must be constant within each stratum. (optional)
`strata_n`	Use for a design in which the numeric vector `strata_n` gives the number of units to sample within each stratum. Must be as long as the number of strata, in the same order as `sort(unique(strata))`. (optional)
`strata_prob`	Use for a design in which `strata_prob` gives the probability of being sampled within each stratum. Must be in the same order as `sort(unique(strata))`. Differs from `prob` in that the probability of being sampled can vary across strata. (optional)
`check_inputs`	Logical. Whether to verify before sampling that the arguments are internally consistent: that counts do not exceed the stratum sizes, that probabilities lie between 0 and 1, that stratum-level arguments have one entry per stratum, and so on. Defaults to `TRUE`. Set to `FALSE` to skip the checks when drawing many samples from arguments that have already been verified; declaring the design once with `declare_rs()` and drawing from it with `draw_rs()` does this for you. (optional)

Details

These are the quantities inverse-probability weights are built from: weight each sampled unit by the reciprocal of its inclusion probability, which `obtain_inclusion_probabilities()` extracts for you.

Value

A numeric vector of length N giving each unit's probability of being included in the sample.

See Also

`strata_rs()`

Examples

```
strata <- rep(c("A", "B", "C"), times = c(50, 100, 200))

probs <- strata_rs_probabilities(strata = strata)
table(strata, probs)

probs <- strata_rs_probabilities(strata = strata, prob = 0.2)
table(strata, probs)

probs <- strata_rs_probabilities(strata = strata, strata_prob = c(0.1, 0.2, 0.3))
table(strata, probs)

probs <- strata_rs_probabilities(strata = strata, strata_n = c(10, 40, 70))
table(strata, probs)
```

Index