# Package 'nullcat'

December 18, 2025

**Type** Package

**Title** Null Models for Categorical and Continuous Community Matrices

**Version** 0.1.0

**Description** Provides null model algorithms for categorical and quantitative community ecology data. Extends classic binary null models (e.g., 'curveball', 'swap') to work with categorical data. Provides a stratified randomization framework for continuous data.

**License** MIT + file LICENSE

**Encoding** UTF-8

**RoxygenNote** 7.3.2

**LinkingTo** Rcpp

**Imports** Rcpp

**Suggests** knitr, rmarkdown, testthat (>= 3.0.0), vegan

**Config/testthat/edition** 3

**URL** https://github.com/matthewkling/nullcat, https://matthewkling.github.io/nullcat/

**BugReports** https://github.com/matthewkling/nullcat/issues

**VignetteBuilder** knitr

**NeedsCompilation** yes

**Author** Matthew Kling [aut, cre, cph]

**Maintainer** Matthew Kling <mattkling@berkeley.edu>

**Repository** CRAN

**Date/Publication** 2025-12-18 14:30:07 UTC

## Contents

---

`c0cat` *Column-constrained categorical randomization (c0cat)*

---

## Description

`c0cat()` preserves the multiset of categories within each column but randomizes their positions across rows, leaving row margins free. This is the categorical analog to vegan's `c0` algorithm. It is a non-sequential method.

## Usage

```
c0cat(x, n_iter = 1L, output = c("category", "index"), seed = NULL)
```

## Arguments

| | |
|---|---|
| `x` | A matrix of categorical data, encoded as integers. Values should represent category or stratum membership for each cell. |
| `n_iter` | Number of iterations. Default is 1000. Larger values yield more thorough mixing. Ignored for non-sequential methods. Minimum burn-in times can be estimated with `suggest_n_iter()`. |
| `output` | Character indicating type of result to return:<br>• `"category"` (default) returns randomized matrix<br>• `"index"` returns an index matrix describing where original entries (a.k.a. "tokens") moved. Useful mainly for testing, and for applications like `quantize()` that care about token tracking in addition to generic integer categories. |
| `seed` | Integer used to seed random number generator, for reproducibility. |

## Value

A matrix of the same dimensions as x, either randomized categorical values (when output = "category") or an integer index matrix describing the permutation of entries (when output = "index").

## Examples

```
set.seed(123)
x <- matrix(sample(1:4, 100, replace = TRUE), nrow = 10)

# Randomize within columns (column margins fixed, row margins free)
x_rand <- c0cat(x)

# Verify columns are preserved but rows are not
all.equal(sort(x[, 1]), sort(x_rand[, 1]))
any(sort(x[1, ]) != sort(x_rand[1, ]))
```

---

# curvecat *Categorical curveball randomization (curvecat)*

---

## Description

Categorical generalization of the binary curveball algorithm (Strona et al. 2014) to matrices of categorical data. This function is a convenience wrapper around nullcat() with method = "curvecat".

## Usage

```
curvecat(x, n_iter = 1000L, output = "category", swaps = "auto", seed = NULL)
```

## Arguments

| | |
|---|---|
| x | A matrix of categorical data, encoded as integers. Values should represent category or stratum membership for each cell. |
| n_iter | Number of iterations. Default is 1000. Larger values yield more thorough mixing. Ignored for non-sequential methods. Minimum burn-in times can be estimated with suggest_n_iter(). |
| output | Character indicating type of result to return:<br>• "category" (default) returns randomized matrix<br>• "index" returns an index matrix describing where original entries (a.k.a. "tokens") moved. Useful mainly for testing, and for applications like quantize() that care about token tracking in addition to generic integer categories. |
| swaps | Character string controlling the direction of token movement. Only used when method is "curvecat", "swapcat", or "tswapcat". Affects the result only when output = "index", otherwise it only affects computation speed. Options include:<br>• "vertical": Tokens move between rows (stay within columns). |

- "horizontal": Tokens move between columns (stay within rows).
- "alternating": Tokens move in both dimensions, alternating between vertical and horizontal swaps. Provides full 2D mixing without preserving either row or column token sets.
- "auto" (default): For output = "category", automatically selects the fastest option based on matrix dimensions. For output = "index", defaults to "alternating" for full mixing.

seed Integer used to seed random number generator, for reproducibility.

## Details

The curvecat algorithm randomizes a categorical matrix while keeping the category multisets of each row and column fixed. In other words, the permuted matrix has the same set of integer values in every row and every column as the original matrix, but they are permuted. It operates on pairs of rows at a time, grouping differing entries by unordered category pairs and redistributing the orientation of those pairs while preservingn the multiset of categories within each row. When there are only two categories, curvecat() reduces to the behavior of the original binary curveball algorithm (Strona et al. 2014) applied to a 0/1 matrix.

## Value

A matrix of the same dimensions as x, either randomized categorical values (when output = "category") or an integer index matrix describing the permutation of entries (when output = "index").

## References

Strona, G., Nappo, D., Boccacci, F., Fattorini, S., & San-Miguel-Ayanz, J. (2014). A fast and unbiased procedure to randomize ecological binary matrices with fixed row and column totals. *Nature Communications*, 5, 4114.

## See Also

nullcat(), nullcat_methods()

## Examples

```
# Create a categorical matrix
set.seed(123)
x <- matrix(sample(1:4, 100, replace = TRUE), nrow = 10)

# Randomize preserving row and column category multisets
x_rand <- curvecat(x, n_iter = 1000)

# Verify margins are preserved
all.equal(sort(x[1, ]), sort(x_rand[1, ])) # row multisets preserved
all.equal(sort(x[, 1]), sort(x_rand[, 1])) # column multisets preserved

# Use with a seed for reproducibility
x_rand1 <- curvecat(x, n_iter = 1000, seed = 42)
x_rand2 <- curvecat(x, n_iter = 1000, seed = 42)
```

```
identical(x_rand1, x_rand2)
```

---

`nullcat` *Categorical matrix randomization*

---

## Description

Categorical generalizations of binary community null model algorithms.

## Usage

```
nullcat(
  x,
  method = nullcat_methods(),
  n_iter = 1000L,
  output = c("category", "index"),
  swaps = c("auto", "vertical", "horizontal", "alternating"),
  seed = NULL
)
```

## Arguments

| | |
|---|---|
| `x` | A matrix of categorical data, encoded as integers. Values should represent category or stratum membership for each cell. |
| `method` | Character specifying the randomization algorithm to use. Options include the following; see details and linked functions for more info.<br>• `"curvecat"`: categorical analog to `curveball`; see `curvecat()` for details.<br>• `"swapcat"`: categorical analog to `swap`; see `swapcat()` for details.<br>• `"tswapcat"`: categorical analog to `tswap`; see `tswapcat()` for details.<br>• `"r0cat"`: categorical analog to `r0`; see `r0cat()` for details.<br>• `"c0cat"`: categorical analog to `c0`; see `c0cat()` for details. |
| `n_iter` | Number of iterations. Default is 1000. Larger values yield more thorough mixing. Ignored for non-sequential methods. Minimum burn-in times can be estimated with `suggest_n_iter()`. |
| `output` | Character indicating type of result to return:<br>• `"category"` (default) returns randomized matrix<br>• `"index"` returns an index matrix describing where original entries (a.k.a. "tokens") moved. Useful mainly for testing, and for applications like `quantize()` that care about token tracking in addition to generic integer categories. |
| `swaps` | Character string controlling the direction of token movement. Only used when method is `"curvecat"`, `"swapcat"`, or `"tswapcat"`. Affects the result only when `output = "index"`, otherwise it only affects computation speed. Options include: |

- `"vertical"`: Tokens move between rows (stay within columns).
- `"horizontal"`: Tokens move between columns (stay within rows).
- `"alternating"`: Tokens move in both dimensions, alternating between vertical and horizontal swaps. Provides full 2D mixing without preserving either row or column token sets.
- `"auto"` (default): For `output = "category"`, automatically selects the fastest option based on matrix dimensions. For `output = "index"`, defaults to `"alternating"` for full mixing.

`seed` Integer used to seed random number generator, for reproducibility.

## Details

`curvecat`, `swapcat`, and `tswapcat` are sequential algorithms that hold category multisets fixed in every row and column. These three algorithms typically reach the same stationary distribution. They differ primarily in efficiency, with `curvecat` being the most efficient (i.e. fewest steps to become fully mixed); `swapcat` and `tswapcat` are thus useful mainly for methodological comparison.

The `r0cat` algorithm holds category multisets fixed in rows but not columns, while `c0cat` does the opposite.

Note that categorical null models are for cell-level categorical data. Site-level attributes (e.g., land cover) or species-level attributes (e.g., functional traits) should be analyzed using different approaches.

## Value

A matrix of the same dimensions as `x`, either randomized categorical values (when `output = "category"`) or an integer index matrix describing the permutation of entries (when `output = "index"`).

## See Also

`nullcat_batch()` for efficient generation of multiple randomized matrices; `nullcat_commsim()` for integration with `vegan`.

## Examples

```
# Create a categorical matrix
set.seed(123)
x <- matrix(sample(1:4, 100, replace = TRUE), nrow = 10)

# Randomize using curvecat method (preserves row & column margins)
x_rand <- nullcat(x, method = "curvecat", n_iter = 1000)

# Check that row multisets are preserved
all.equal(sort(x[1, ]), sort(x_rand[1, ]))

# Get index output showing where each cell moved
idx <- nullcat(x, method = "curvecat", n_iter = 1000, output = "index")

# Use different methods
x_swap <- nullcat(x, method = "swapcat", n_iter = 1000)
```

```
x_r0 <- nullcat(x, method = "r0cat")

# Use with a seed for reproducibility
x_rand1 <- nullcat(x, method = "curvecat", n_iter = 1000, seed = 42)
x_rand2 <- nullcat(x, method = "curvecat", n_iter = 1000, seed = 42)
identical(x_rand1, x_rand2)
```

---

`nullcat_batch` *Generate a batch of null matrices using nullcat()*

---

## Description

Runs the categorical null model implemented in `nullcat()` repeatedly, generating a batch of randomized matrices or, optionally, a batch of summary statistics computed from those matrices. This is the categorical analog of `quantize_batch()`.

## Usage

```
nullcat_batch(x, n_reps = 999L, stat = NULL, n_cores = 1L, seed = NULL, ...)
```

## Arguments

| | |
|---|---|
| `x` | Community matrix (sites × species) or any categorical matrix of integers. |
| `n_reps` | Number of randomizations to generate. Default is `999`. |
| `stat` | Optional summary function taking a matrix and returning a numeric statistic. If `NULL` (default), the function returns the full set of randomized matrices. |
| `n_cores` | Number of compute cores to use for parallel processing. Default is `1`. |
| `seed` | Integer used to seed random number generator, for reproducibility. |
| `...` | Additional arguments passed to `nullcat()` (e.g. `method`, `n_iter`, `output`). |

## Value

If `stat` is `NULL`, returns a 3D array (rows × cols × n_reps). If `stat` is not `NULL`, returns a numeric array of statistic values (dimensionality depends on `stat`).

## Examples

```
set.seed(123)
x <- matrix(sample(1:4, 100, replace = TRUE), nrow = 10)

# Generate 99 randomized matrices
nulls <- nullcat_batch(x, n_reps = 99, method = "curvecat", n_iter = 100)

# Or compute a statistic on each
row_sums <- nullcat_batch(x, n_reps = 99, stat = rowSums,
                          method = "curvecat", n_iter = 100)
```

```
# Specify multiple cores for parallel processing
nulls <- nullcat_batch(x, n_reps = 99, n_iter = 100, n_cores = 2)
```

---

`nullcat_commsim` *Nullcat-based commsim (non-sequential)*

---

## Description

Construct a `vegan::commsim()` object that uses `nullcat()` as a non-sequential null model for categorical / integer matrices. Each simulated matrix is generated independently by applying `nullcat()` with `n_iter` trades starting from the original matrix.

## Usage

```
nullcat_commsim(
  n_iter = 10000,
  method = nullcat_methods(),
  output = c("category", "index")
)
```

## Arguments

| | |
|---|---|
| `n_iter` | Integer, number of iterations (trades) per simulated matrix. Must be a positive integer. Default is `1e4`. |
| `method` | Character specifying which nullcat randomization algorithm to use. See `nullcat()` and `nullcat_methods()` for details. |
| `output` | Character, passed to `nullcat(output = ...)`. Typically `"category"` (default) or `"index"`. |

## Value

An object of class `"commsim"` suitable for use with `vegan::nullmodel()` and `vegan::oecosimu()`.

## Details

This generates a commsim object that is **non-sequential**: each simulated matrix starts from the original matrix and is randomized independently using `n_iter` trades of the chosen `method`.

When used via `vegan::simulate.nullmodel()`, the arguments behave as:

- `nsim`: number of simulated matrices to generate.
- `n_iter` (here, in `nullcat_commsim()`): number of trades per simulated matrix (controls how strongly each replicate is shuffled).
- `burnin` and `thin`: are **ignored** for this commsim, because `isSeq = FALSE` (the simulations are not a Markov chain).

In other words, treat `n_iter` as the tuning parameter for how thoroughly each independent null matrix is randomized.

## See Also

`nullcat_batch()` if you just want a batch of null matrices without going through **vegan**.

## Examples

```
library(vegan)

x  <- matrix(sample(1:5, 50, replace = TRUE), 10, 5)
cs <- nullcat_commsim(n_iter = 1e4, method = "curvecat")

nm   <- nullmodel(x, cs)
sims <- simulate(nm, nsim = 999)
```

---

`nullcat_commsim_seq` *Nullcat-based commsim (sequential / Markov chain)*

---

## Description

Construct a `vegan::commsim()` object that uses `nullcat()` as a *sequential* null model: successive simulated matrices form a Markov chain. Internally, each simulation "step" advances the chain by `thin` trades of the chosen `method` (e.g. `"curvecat"`), where `thin` is supplied via 'vegan::simulate.nullmodel()" arguments. This is analogous to how sequential swap / curveball null models are used in **vegan**, but extended to categorical data via `nullcat()`.

## Usage

```
nullcat_commsim_seq(
  method = nullcat_methods(),
  output = c("category", "index")
)
```

## Arguments

| | |
|---|---|
| `method` | Character specifying which nullcat randomization algorithm to use. See `nullcat()` and `nullcat_methods()` for details. |
| `output` | Character, passed to `nullcat(output = ...)`. Typically `"category"` (default) or `"index"`. |

## Value

An object of class `"commsim"` suitable for use with `vegan::nullmodel()` and `vegan::oecosimu()`.

## Details

This model is **sequential**: simulated matrices form a Markov chain. The current matrix is updated in-place by repeated calls to the randomization model, and successive matrices are obtained by advancing the chain.

In `vegan::simulate.nullmodel()`, the control arguments behave as:

- `nsim`: number of matrices to *store* from the chain.
- `thin`: number of trades per step. Each "step" of the chain applies `thin` trades of the chosen `method` to the current state before possibly storing it.
- `burnin`: number of initial steps to perform (each with `thin` trades) before storing any matrices, i.e. the Markov chain burn-in.

There is no `n_iter` argument here: mixing is controlled entirely by `thin` (trades per step) and `burnin` (number of initial steps discarded), in the same spirit as sequential swap / curveball models in **vegan**.

## Examples

```
library(vegan)

x  <- matrix(sample(1:5, 50, replace = TRUE), 10, 5)
cs <- nullcat_commsim_seq(method = "curvecat")

nm <- nullmodel(x, cs)

# control the chain with 'thin' and 'burnin'
sims <- simulate(nm, nsim = 999, thin = 100, burnin = 1000)
```

---

`nullcat_methods` *Supported nullcat methods*

---

## Description

Return the character vector of supported categorical randomization methods.

## Usage

```
nullcat_methods()
```

## Value

A character vector of method names.

## Examples

```
nullcat_methods()
```

quantize *Stratified randomization of a quantitative community matrix*

## Description

quantize() is a community null model for quantitative community data (e.g. abundance, biomass, or occurrence probability). It works by converting quantitative values into discrete strata, randomizing the stratified matrix using a categorical null model, and reassigning quantitative values within strata according to a specified constraint.

## Usage

```
quantize(
  x = NULL,
  prep = NULL,
  method = nullcat_methods(),
  fixed = c("cell", "stratum", "row", "col"),
  breaks = NULL,
  n_strata = 5,
  transform = identity,
  offset = 0,
  zero_stratum = FALSE,
  n_iter = 1000,
  seed = NULL
)
```

## Arguments

| | |
|---|---|
| x | Community matrix with sites in rows, species in columns, and nonnegative quantitative values in cells. Ignored if prep is supplied. |
| prep | Optional precomputed object returned by quantize_prep(). If supplied, x is ignored and all overhead (stratification, pools, etc.) is taken from prep, which is typically much faster when generating many randomizations of the same dataset. |
| method | Character string specifying the null model algorithm. The default "curvecat" uses the categorical curveball algorithm. See nullcat() for alternative options. |
| fixed | Character string specifying the level at which quantitative values are held fixed during randomization. One of:<br>• "cell" (the default; only available when method = "curvecat"): values remain attached to their original cells and move with them during the categorical randomization. Row and column value distributions are not preserved, but the mapping between each original cell and its randomized destination is fixed.<br>• "stratum": values are shuffled globally within each stratum, holding only the overall stratum-level value distribution fixed.<br>• "row": values are shuffled within strata separately for each row, holding each row's value multiset fixed. Not compatible with all methods. |

- `"col"`: values are shuffled within strata separately for each column, holding each column's value multiset fixed.

Note that this interacts with `method`: different null models fix different margins in the underlying binary representation.

`breaks` Numeric vector of stratum breakpoints.

`n_strata` Integer giving the number of strata to split the data into. Must be 2 or greater. Larger values yield randomizations with less mixing but higher fidelity to the original marginal distributions. Default is 5. Ignored unless `breaks = NULL`.

`transform` A function used to transform the values in `x` before assigning them to `n_strata` equal-width intervals. Examples include `sqrt`, `log`, `rank` (for equal-occupancy strata), etc.; the default is `identity`. If `zero_stratum = TRUE`, the transformation is only applied on nonzero values. The function should pass NA values. This argument is ignored unless `breaks = NULL`.

`offset` Numeric value between -1 and 1 (default 0) indicating how much to shift stratum breakpoints relative to the binwidth (applied during quantization as: `breaks <- breaks + offset * bw`). To assess sensitivity to stratum boundaries, run `quantize()` multiple times with different offset values. Ignored unless `breaks = NULL`.

`zero_stratum` Logical indicating whether to segregate zeros into their own stratum. If `FALSE` (the default), zeros will likely be combined into a stratum that also includes small positive numbers. If `breaks` is specified, zero simply gets added as an additional break; if not, one of the `n_strata` will represent zeros and the others will be nonzero ranges.

`n_iter` Number of iterations. Default is 1000. Larger values yield more thorough mixing. Ignored for non-sequential methods. Minimum burn-in times can be estimated with `suggest_n_iter()`.

`seed` Integer used to seed random number generator, for reproducibility.

## Details

This approach provides a framework for preserving row and/or column value distributions in continuous data. When using `fixed = "row"` or `fixed = "col"`, one dimension's value multisets are preserved exactly while the other is preserved at the resolution of strata, approximating a fixed-fixed null model for quantitative data. The number of strata controls the tradeoff between preservation fidelity and randomization strength.

By default, `quantize()` will compute all necessary overhead for a given dataset (strata, pools, etc.) internally. For repeated randomization of the same matrix (e.g. to build a null distribution), this overhead can be computed once using `quantize_prep()` and reused by supplying the resulting object via the `prep` argument.

## Value

A randomized version of `x`, with the same dimensions and dimnames. For `method = "curvecat"`, the quantitative values are reassigned within strata while preserving row and column stratum multisets. For binary methods, the result corresponds to applying the chosen binary null model to each stratum and recombining.

## See Also

quantize_batch() for efficient generation of multiple randomized matrices; quantize_commsim() for integration with vegan.

## Examples

```
# toy quantitative community matrix
set.seed(1)
comm <- matrix(rexp(50 * 40), nrow = 50,
               dimnames = list(paste0("site", 1:50),
                               paste0("sp", 1:40)))

# default: curvecat-backed stratified randomization
rand1 <- quantize(comm)

# change stratification and preservation mode
rand2 <- quantize(comm, n_strata = 4,
                  transform = sqrt,
                  fixed  = "row",
                  n_iter    = 2000)

# use a different randomization algorithm
rand3 <- quantize(comm, method = "swapcat", n_iter = 10000)

# precompute overhead and reuse for many randomizations
prep  <- quantize_prep(comm, method = "curvecat",
                       n_strata = 5, fixed = "row")
rand4 <- quantize(prep = prep)
rand5 <- quantize(prep = prep)
```

---

# quantize_batch *Generate a batch of null matrices using quantize()*

---

## Description

Runs the stratified null model implemented in quantize() repeatedly, generating a batch of randomized matrices or, optionally, a batch of summary statistics computed from those matrices.

## Usage

```
quantize_batch(x, n_reps = 999L, stat = NULL, n_cores = 1L, seed = NULL, ...)
```

## Arguments

| | |
|---|---|
| x | Community matrix (species × sites, or any numeric matrix). |
| n_reps | Number of randomizations to generate. Default is 999. |

| | |
|---|---|
| stat | Optional summary function taking a matrix and returning a numeric statistic (e.g. rowSums with abundance data would give total abundance per site). If NULL (default), the function returns the full set of randomized matrices. |
| n_cores | Number of compute cores to use for parallel processing. Default is 1. |
| seed | Integer used to seed random number generator, for reproducibility. |
| ... | Additional arguments passed to quantize(), (e.g. method, breaks, n_strata, transform, offset, zero_stratum, fixed, n_iter, etc.). |

## Value

If stat is NULL, returns a 3D array (rows × cols × n_reps). If stat is not NULL, returns a numeric array of statistic values (dimensionality depends on stat).

## Examples

```
set.seed(123)
x <- matrix(runif(100), nrow = 10)

# Generate 99 randomized matrices
nulls <- quantize_batch(x, n_reps = 99, method = "curvecat", n_iter = 100)

# Or compute a statistic on each
row_sums <- nullcat_batch(x, n_reps = 99, stat = rowSums,
                          method = "curvecat", n_iter = 100)

# Specify multiple cores for parallel processing
nulls <- quantize_batch(x, n_reps = 99, n_iter = 100, n_cores = 2)
```

---

# quantize_commsim *Quantize-based commsim (non-sequential)*

---

## Description

Construct a vegan::commsim() object that uses quantize() as a non-sequential null model for numeric community matrices. Each simulated matrix is generated independently by applying quantize() with n_iter trades (via its internal call to nullcat()) starting from the original matrix.

## Usage

```
quantize_commsim(n_iter = 10000, ...)
```

## Arguments

| | |
|---|---|
| n_iter | Integer, number of iterations (trades) per simulated matrix. Must be a positive integer. Default is 1e4. |
| ... | Arguments passed to quantize(), such as breaks, n_strata, transform, offset, zero_stratum, fixed, method, etc. Do **not** supply x or n_iter here; these are set internally by quantize_commsim(). See quantize() for details. |

## Value

An object of class "commsim" suitable for vegan::nullmodel() and vegan::oecosimu().

## Details

This generates a commsim object that is **non-sequential**: each simulated matrix starts from the original matrix and is randomized independently using n_iter trades of the chosen method.

When used via vegan::simulate.nullmodel(), the arguments behave as:

- nsim: number of simulated matrices to generate.
- n_iter (here, in nullcat_commsim()): number of trades per simulated matrix (controls how strongly each replicate is shuffled).
- burnin and thin: are **ignored** for this commsim, because isSeq = FALSE (the simulations are not a Markov chain).

In other words, treat n_iter as the tuning parameter for how thoroughly each independent null matrix is randomized.

## See Also

quantize_batch() if you just want a batch of null matrices without going through **vegan**.

## Examples

```
library(vegan)

x <- matrix(rexp(50), 10, 5)

cs <- quantize_commsim(
  n_strata = 10,
  method   = "curvecat",
  n_iter   = 1000L
)

nm   <- nullmodel(x, cs)
sims <- simulate(nm, nsim = 999)
```

`quantize_commsim_seq` *Quantile-based quantize commsim (sequential / Markov chain)*

## Description

Construct a `vegan::commsim()` object that uses `quantize()` as a *sequential* null model: successive simulated matrices form a Markov chain in the space of numeric community matrices. Internally, each simulation "step" advances the chain by re-applying `quantize()` to the current matrix using the settings provided via `...`.

## Usage

```
quantize_commsim_seq(...)
```

## Arguments

| | |
|---|---|
| `...` | Arguments passed to `quantize()`, such as `breaks`, `n_strata`, `transform`, `offset`, `zero_stratum`, `fixed`, `method`, `n_iter`, etc. Do **not** supply `x` or `n_iter` here; `x` is provided by vegan and `n_iter` is set internally from `thin`. See `quantize()` for details. |

## Value

An object of class `"commsim"` suitable for `vegan::nullmodel()` and `vegan::oecosimu()`.

## Details

This model is **sequential**: simulated matrices form a Markov chain. The current matrix is updated in-place by repeated calls to the randomization model, and successive matrices are obtained by advancing the chain.

In `vegan::simulate.nullmodel()`, the control arguments behave as:

- `nsim`: number of matrices to *store* from the chain.
- `thin`: number of trades per step. Each "step" of the chain applies `thin` trades of the chosen `method` to the current state before possibly storing it.
- `burnin`: number of initial steps to perform (each with `thin` trades) before storing any matrices, i.e. the Markov chain burn-in.

There is no `n_iter` argument here: mixing is controlled entirely by `thin` (trades per step) and `burnin` (number of initial steps discarded), in the same spirit as sequential swap / curveball models in **vegan**.

## Examples

```
library(vegan)

x <- matrix(rexp(50), 10, 5)

cs <- quantize_commsim_seq(
  n_strata = 5,
  method   = "curvecat"
)

nm <- nullmodel(x, cs)

sims <- simulate(
  nm,
  nsim   = 999,
  thin   = 10,    # 10 quantize updates between stored states
  burnin = 100    # 100 initial steps discarded
)
```

---

quantize_prep *Prepare stratified null model overhead for quantize()*

---

## Description

quantize_prep() precomputes all of the stratification and bookkeeping needed by quantize() for a given quantitative community matrix. This is useful when you want to generate many randomizations of the same dataset: the expensive steps (strata assignment, value pools, and arguments for the underlying null model) are computed once, and the resulting object can be passed to quantize(prep = ...) for fast repeated draws.

## Usage

```
quantize_prep(
  x,
  method = nullcat_methods(),
  fixed = c("cell", "stratum", "row", "col"),
  breaks = NULL,
  n_strata = 5,
  transform = identity,
  offset = 0,
  zero_stratum = FALSE,
  n_iter = 1000
)
```

## Arguments

| | |
|---|---|
| x | Community matrix with sites in rows, species in columns, and nonnegative quantitative values in cells. This is the dataset for which stratification and null model overhead should be prepared. |
| method | Character string specifying the null model algorithm. The default "curvecat" uses the categorical curveball algorithm. See nullcat() for alternative options. |
| fixed | Character string specifying the level at which quantitative values are held fixed during randomization. One of:<br>• "cell" (the default; only available when method = "curvecat"): values remain attached to their original cells and move with them during the categorical randomization. Row and column value distributions are not preserved, but the mapping between each original cell and its randomized destination is fixed.<br>• "stratum": values are shuffled globally within each stratum, holding only the overall stratum-level value distribution fixed.<br>• "row": values are shuffled within strata separately for each row, holding each row's value multiset fixed. Not compatible with all methods.<br>• "col": values are shuffled within strata separately for each column, holding each column's value multiset fixed.<br>Note that this interacts with method: different null models fix different margins in the underlying binary representation. |
| breaks | Numeric vector of stratum breakpoints. |
| n_strata | Integer giving the number of strata to split the data into. Must be 2 or greater. Larger values yield randomizations with less mixing but higher fidelity to the original marginal distributions. Default is 5. Ignored unless breaks = NULL. |
| transform | A function used to transform the values in x before assigning them to n_strata equal-width intervals. Examples include sqrt, log, rank (for equal-occupancy strata), etc.; the default is identity. If zero_stratum = TRUE, the transformation is only applied on nonzero values. The function should pass NA values. This argument is ignored unless breaks = NULL. |
| offset | Numeric value between -1 and 1 (default 0) indicating how much to shift stratum breakpoints relative to the binwidth (applied during quantization as: breaks <- breaks + offset * bw). To assess sensitivity to stratum boundaries, run quantize() multiple times with different offset values. Ignored unless breaks = NULL. |
| zero_stratum | Logical indicating whether to segregate zeros into their own stratum. If FALSE (the default), zeros will likely be combined into a stratum that also includes small positive numbers. If breaks is specified, zero simply gets added as an additional break; if not, one of the n_strata will represent zeros and the others will be nonzero ranges. |
| n_iter | Number of iterations. Default is 1000. Larger values yield more thorough mixing. Ignored for non-sequential methods. Minimum burn-in times can be estimated with suggest_n_iter(). |

## Details

Internally, quantize_prep():

- transforms and stratifies x into n_strata numeric intervals (via stratify()),
- constructs the appropriate value pools given fixed, and
- assembles arguments for the underlying null model call to nullcat().

The returned object can be reused across calls to quantize(), quantize_batch(), or other helpers that accept a prep argument.

## Value

A list with class "quantize_prep" (if you want to set it) containing the components needed by quantize():

- x: original quantitative marix x,
- strata: integer matrix of the same dimension as x, giving the stratum index (1:n_strata) for each cell.
- pool: data structure encoding the quantitative value pools used during reassignment.
- method: the null model method used (as in the method argument).
- n_strata, transform, offset, fixed: the stratification and reassignment settings used to construct strata and pool.
- sim_args: named list of arguments passed to nullcat() (e.g. n_iter).

This object is intended to be passed unchanged to the prep argument of quantize() or quantize_batch().

## See Also

quantize(), quantize_batch()

## Examples

```
set.seed(1)
comm <- matrix(rexp(50 * 40), nrow = 50,
               dimnames = list(paste0("site", 1:50),
                               paste0("sp", 1:40)))

# prepare overhead for a curvecat-backed stratified null model
prep <- quantize_prep(comm, method = "curvecat",
                      n_strata = 5,
                      fixed = "row",
                      n_iter = 2000)

# fast repeated randomizations using the same prep
rand1 <- quantize(prep = prep)
rand2 <- quantize(prep = prep)
```

r0cat *Row-constrained categorical randomization (r0cat)*

## Description

r0cat() preserves the multiset of categories within each row but randomizes their positions across columns, leaving column margins free. This is the categorical analog to vegan's r0 algorithm. It is a non-sequential method.

## Usage

```
r0cat(x, n_iter = 1L, output = c("category", "index"), seed = NULL)
```

## Arguments

| | |
|---|---|
| x | A matrix of categorical data, encoded as integers. Values should represent category or stratum membership for each cell. |
| n_iter | Number of iterations. Default is 1000. Larger values yield more thorough mixing. Ignored for non-sequential methods. Minimum burn-in times can be estimated with suggest_n_iter(). |
| output | Character indicating type of result to return:<br>• "category" (default) returns randomized matrix<br>• "index" returns an index matrix describing where original entries (a.k.a. "tokens") moved. Useful mainly for testing, and for applications like quantize() that care about token tracking in addition to generic integer categories. |
| seed | Integer used to seed random number generator, for reproducibility. |

## Value

A matrix of the same dimensions as x, either randomized categorical values (when output = "category") or an integer index matrix describing the permutation of entries (when output = "index").

## Examples

```
set.seed(123)
x <- matrix(sample(1:4, 100, replace = TRUE), nrow = 10)

# Randomize within rows (row margins fixed, column margins free)
x_rand <- r0cat(x)

# Verify rows are preserved but columns are not
all.equal(sort(x[1, ]), sort(x_rand[1, ]))
any(sort(x[, 1]) != sort(x_rand[, 1]))
```

`stratify` *Bin quantitative data into strata*

## Description

Bin quantitative data into strata

## Usage

```
stratify(
  x,
  breaks = NULL,
  n_strata = 5,
  transform = identity,
  offset = 0,
  zero_stratum = FALSE
)
```

## Arguments

| | |
|---|---|
| `x` | A matrix or vector containing non-negative values. |
| `breaks` | Numeric vector of stratum breakpoints. |
| `n_strata` | Integer giving the number of strata to split the data into. Must be 2 or greater. Larger values yield randomizations with less mixing but higher fidelity to the original marginal distributions. Default is 5. Ignored unless `breaks = NULL`. |
| `transform` | A function used to transform the values in `x` before assigning them to `n_strata` equal-width intervals. Examples include `sqrt`, `log`, `rank` (for equal-occupancy strata), etc.; the default is `identity`. If `zero_stratum = TRUE`, the transformation is only applied on nonzero values. The function should pass NA values. This argument is ignored unless `breaks = NULL`. |
| `offset` | Numeric value between -1 and 1 (default 0) indicating how much to shift stratum breakpoints relative to the binwidth (applied during quantization as: `breaks <- breaks + offset * bw`). To assess sensitivity to stratum boundaries, run `quantize()` multiple times with different offset values. Ignored unless `breaks = NULL`. |
| `zero_stratum` | Logical indicating whether to segregate zeros into their own stratum. If `FALSE` (the default), zeros will likely be combined into a stratum that also includes small positive numbers. If `breaks` is specified, zero simply gets added as an additional break; if not, one of the `n_strata` will represent zeros and the others will be nonzero ranges. |

## Value

An object the same size as x, with integer values representing stratum classifications.

## Examples

```
# Stratify a numeric vector
x <- c(0, 0, 0.1, 0.5, 1.2, 3.4, 5.6, 10.2)
stratify(x, n_strata = 3)

# With transformation
stratify(x, n_strata = 3, transform = log1p)

# Separate zero stratum
stratify(x, n_strata = 3, zero_stratum = TRUE)
```

---

`suggest_n_iter` *Suggest a reasonable n_iter for a randomization*

---

## Description

Uses trace diagnostics to estimate how many burn-in iterations are needed for a `nullcat` or `quantize` randomization to reach its apparent stationary distribution, given a dataset and randomization method. Uses a "first pre-tail sign-crossing" rule per chain, then returns the maximum across chains. Can be called on a community matrix or a `cat_trace` object.

## Usage

```
suggest_n_iter(trace = NULL, tail_frac = 0.3, plot = FALSE, ...)
```

## Arguments

| | |
|---|---|
| `trace` | Either a `cat_trace` object (as returned by `trace_cat()`), or NULL. If NULL, arguments to `trace_cat()`, including `x` and any other relevant parameters must be supplied via `...` |
| `tail_frac` | Fraction of the trace (at the end) used as the tail window (default 0.3). |
| `plot` | If TRUE, plot the trace, with a vertical line at the suggested value. |
| `...` | Arguments passed to `trace_cat()` including arguments it passes to the `nullcat()` or `quantize()` function. Ignored if `trace` is non-NULL. |

## Details

This function uses a “first pre-tail sign-crossing” heuristic to identify burn-in cutoff. This is a simple variant of standard mean-stability tests used in MCMC convergence diagnostics (e.g., Heidelberger & Welch 1983; Geweke 1992; Geyer 1992). It computes the long-run mean based on the "tail window" of the chain, and detects the first iteration at which the trace statistic crosses this long-run mean, indicating that the chain has begun to oscillate around its stationary value. If the chain does not reach the long-run mean before the start of the tail window, the chain is determined not to have reached stationarity, and the function returns `NA` with attribute `converged = FALSE`.

## Value

An integer of class "nullcat_n_iter" with attributes: n_iter (numeric or NA), trace (matrix), steps (vector), tail_mean (per-chain), per_chain (data.frame), converged (logical).

## References

Heidelberger, P. & Welch, P.D. (1983). Simulation run length control in the presence of an initial transient. Operations Research, 31(6): 1109–1144.

Geweke, J. (1992). Evaluating the accuracy of sampling-based approaches to the calculation of posterior moments. In Bayesian Statistics 4, pp. 169–193.

Geyer, C.J. (1992). Practical Markov Chain Monte Carlo. Statistical Science, 7(4): 473–483.

Feller, W. (1968). An Introduction to Probability Theory and Its Applications, Vol. I. Wiley.

## Examples

```
set.seed(1234)
x <- matrix(sample(1:5, 2500, replace = TRUE), 50)

# call `trace_cat`, then pass result to `suggest_n_iter`:
trace <- trace_cat(x = x, fun = "nullcat", n_iter = 1000,
                   n_chains = 5, method = "curvecat")
suggest_n_iter(trace, tail_frac = 0.3, plot = TRUE)

# alternatively, supply `trace_cat` arguments directly to `suggest_n_iter`:
x <- matrix(runif(2500), 50)
n_iter <- suggest_n_iter(
    x = x, n_chains = 5, n_iter = 1000, tail_frac = 0.3,
    fun = "quantize", n_strata = 4, fixed = "stratum",
    method = "curvecat", plot = TRUE)
```

---

# swapcat *Categorical swap randomization (swapcat)*

---

## Description

Categorical generalization of the binary 2x2 swap algorithm to matrices of categorical data. This function is a convenience wrapper around nullcat() with method = "swapcat".

## Usage

```
swapcat(
  x,
  n_iter = 1000L,
  output = c("category", "index"),
  swaps = "auto",
  seed = NULL
)
```

## Arguments

| | |
|---|---|
| x | A matrix of categorical data, encoded as integers. Values should represent category or stratum membership for each cell. |
| n_iter | Number of iterations. Default is 1000. Larger values yield more thorough mixing. Ignored for non-sequential methods. Minimum burn-in times can be estimated with `suggest_n_iter()`. |
| output | Character indicating type of result to return:<br>• `"category"` (default) returns randomized matrix<br>• `"index"` returns an index matrix describing where original entries (a.k.a. "tokens") moved. Useful mainly for testing, and for applications like `quantize()` that care about token tracking in addition to generic integer categories. |
| swaps | Character string controlling the direction of token movement. Only used when method is `"curvecat"`, `"swapcat"`, or `"tswapcat"`. Affects the result only when `output = "index"`, otherwise it only affects computation speed. Options include:<br>• `"vertical"`: Tokens move between rows (stay within columns).<br>• `"horizontal"`: Tokens move between columns (stay within rows).<br>• `"alternating"`: Tokens move in both dimensions, alternating between vertical and horizontal swaps. Provides full 2D mixing without preserving either row or column token sets.<br>• `"auto"` (default): For `output = "category"`, automatically selects the fastest option based on matrix dimensions. For `output = "index"`, defaults to `"alternating"` for full mixing. |
| seed | Integer used to seed random number generator, for reproducibility. |

## Details

The swapcat algorithm attempts random 2x2 swaps of the form:

```
a b        b a
b a   <->  a b
```

where $a$ and $b$ are distinct categories. These swaps preserve the multiset of categories in each row and column. With only two categories present, `swapcat()` reduces to the behavior of the standard binary swap algorithm.

## Value

A matrix of the same dimensions as `x`, either randomized categorical values (when `output = "category"`) or an integer index matrix describing the permutation of entries (when `output = "index"`).

## References

Gotelli, N. J. (2000). Null model analysis of species co-occurrence patterns. *Ecology*, 81(9), 2606–2621.

See also Gotelli & Entsminger (2003) *EcoSim: Null models software for ecology* (Version 7.0) for implementation details of the binary swap algorithm.

## See Also

curvecat() for an algorithm that produces equivalent results with better computational efficiency.

## Examples

```
set.seed(123)
x <- matrix(sample(1:4, 100, replace = TRUE), nrow = 10)

# Randomize using swap algorithm
x_rand <- swapcat(x, n_iter = 1000)

# Verify fixed-fixed constraint (row and column margins preserved)
all.equal(sort(x[1, ]), sort(x_rand[1, ]))
all.equal(sort(x[, 1]), sort(x_rand[, 1]))
```

---

# trace_cat *Trace diagnostics for categorical randomizations*

---

## Description

Applies nullcat() or quantize() to a community matrix, recording a summary statistic at each iteration to help assess mixing on a given dataset.

## Usage

```
trace_cat(
  x,
  fun = c("nullcat", "quantize"),
  n_iter = 1000L,
  thin = NULL,
  n_chains = 5L,
  n_cores = 1L,
  stat = NULL,
  seed = NULL,
  plot = FALSE,
  ...
)
```

## Arguments

| | |
|---|---|
| x | Matrix of categorical data (integers) or quantitative values. |
| fun | Which function to trace: "nullcat" or "quantize". |
| n_iter | Total number of update iterations to simulate. Default is 1000. |
| thin | Thinning interval (updates per recorded point). Default ~ n_iter/100. Smaller values increase resolution but increase run time. |

| | |
|---|---|
| n_chains | Number of independent chains to run, to assess consistency (default 5). |
| n_cores | Parallel chains (default 1). |
| stat | Function that compares x to a permuted x_rand to quantify their similarity. Either a function f(x, x_rand) returning a scalar, or NULL. If NULL (the default), traces use Cohen's kappa for nullcat() or Pearson's correlation for quantize(). |
| seed | Optional integer seed for reproducible traces. |
| plot | If TRUE, plot the traces. |
| ... | Arguments to the chosen fun (nullcat() or quantize()), such as method, n_strata, fixed, etc. |

## Value

An object of class "cat_trace" with elements:

- traces: matrix of size (n_steps+1) x n_chains, including iteration 0
- steps: integer vector of iteration numbers (starting at 0)
- fun, n_iter, thin, n_chains, n_cores, stat_name, call
- fun_args: list of the ... used (for reproducibility)

Plotting is available via plot(cat_trace).

## Examples

```
# nullcat trace
set.seed(123)
x <- matrix(sample(1:5, 2500, replace = TRUE), 50)
tr <- trace_cat(x, n_iter = 1000, n_chains = 5, fun = "nullcat",
                method = "curvecat")
plot(tr)

# quantize trace
x <- matrix(runif(2500), 50)
tr <- trace_cat(x, n_iter = 1000, n_chains = 5, fun = "quantize",
                method = "curvecat", n_strata = 3, fixed = "cell")
plot(tr)
```

---

# tswapcat *Trial-swap categorical randomization (tswapcat)*

---

## Description

The trial-swap ("tswap") algorithm is a fixed–fixed randomization that repeatedly attempts random 2×2 swaps until a valid one is found in each iteration, reducing the number of wasted draws compared to the simple swap. tswapcat() extends this logic to categorical matrices.

## Usage

```
tswapcat(
  x,
  n_iter = 1000L,
  output = c("category", "index"),
  swaps = "auto",
  seed = NULL
)
```

## Arguments

| | |
|---|---|
| x | A matrix of categorical data, encoded as integers. Values should represent category or stratum membership for each cell. |
| n_iter | Number of iterations. Default is 1000. Larger values yield more thorough mixing. Ignored for non-sequential methods. Minimum burn-in times can be estimated with `suggest_n_iter()`. |
| output | Character indicating type of result to return:<br>• `"category"` (default) returns randomized matrix<br>• `"index"` returns an index matrix describing where original entries (a.k.a. "tokens") moved. Useful mainly for testing, and for applications like `quantize()` that care about token tracking in addition to generic integer categories. |
| swaps | Character string controlling the direction of token movement. Only used when method is `"curvecat"`, `"swapcat"`, or `"tswapcat"`. Affects the result only when `output = "index"`, otherwise it only affects computation speed. Options include:<br>• `"vertical"`: Tokens move between rows (stay within columns).<br>• `"horizontal"`: Tokens move between columns (stay within rows).<br>• `"alternating"`: Tokens move in both dimensions, alternating between vertical and horizontal swaps. Provides full 2D mixing without preserving either row or column token sets.<br>• `"auto"` (default): For `output = "category"`, automatically selects the fastest option based on matrix dimensions. For `output = "index"`, defaults to `"alternating"` for full mixing. |
| seed | Integer used to seed random number generator, for reproducibility. |

## Value

A matrix of the same dimensions as `x`, either randomized categorical values (when `output = "category"`) or an integer index matrix describing the permutation of entries (when `output = "index"`).

## References

Gotelli, N. J. (2000). Null model analysis of species co-occurrence patterns. *Ecology*, 81(9), 2606–2621.

Miklós, I. & Podani, J. (2004). Randomization of presence–absence matrices: comments and new algorithms. *Ecology*, 85(1), 86–92.

Gotelli, N. J. & Entsminger, G. L. (2003). *EcoSim: Null models software for ecology* (Version 7.0). Acquired Intelligence Inc. & Kesey-Bear, Jericho (VT).

## See Also

curvecat() for an algorithm that produces equivalent results with better computational efficiency.

## Examples

```
set.seed(123)
x <- matrix(sample(1:4, 100, replace = TRUE), nrow = 10)

# Randomize using swap algorithm
x_rand <- tswapcat(x, n_iter = 1000)

# Verify fixed-fixed constraint (row and column margins preserved)
all.equal(sort(x[1, ]), sort(x_rand[1, ]))
all.equal(sort(x[, 1]), sort(x_rand[, 1]))
```

# Index