

Package: Saqlab (via r-universe)

October 8, 2026

Title Simulation and Analysis for Temporal Network Analysis

Version 0.4.1

Description Simulation tools for temporal network analysis (TNA) research. Generates synthetic datasets with known ground-truth parameters for testing and validating statistical methods: Markov chain sequences, latent profiles, latent classes, regression, factor analysis, and sequence cluster structures. All simulation functions return list(data, params) for JSON-serializable access to ground-truth parameters.

License MIT + file LICENSE

URL <https://mohsaqr.github.io/Saqlab/>,
<https://github.com/mohsaqr/Saqlab>

BugReports <https://github.com/mohsaqr/Saqlab/issues>

Encoding UTF-8

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.3

Imports tna, seqHMM, dplyr, tidyr, parallel, future, future.apply, progressr, lhs, igraph, ggplot2

Suggests testthat (>= 3.0.0), knitr, rmarkdown, network, sna, ggbridges, cograph, glmnet, lavaan, lme4, mirt, survival

VignetteBuilder knitr

Config/testthat/edition 3

Config/pak/sysreqs cmake libglpk-dev make libicu-dev libjpeg-dev libpng-dev libxml2-dev libssl-dev libx11-dev zlib1g-dev

Repository <https://mohsaqr.r-universe.dev>

Date/Publication 2026-10-08 18:38:18 UTC

RemoteUrl <https://github.com/mohsaqr/Saqlab>

RemoteRef HEAD

RemoteSha 6aab417a53dd09f9583e19226145a975c52689d4

Contents

batch_apply	3
batch_fit_models	5
compare_centralities	6
compare_edge_recovery	7
compare_estimation	9
compare_network_estimation	11
compare_networks	13
compare_reliability	14
cross_validate_tna	17
export_simulation	18
fit_network_model	19
generate_param_grid	20
generate_probabilities	22
get_global_names	23
get_learning_states	24
get_scenario	25
GLOBAL_NAMES	26
GROUP_REGULATION_ACTIONS	27
inject_missingness	28
learning_states	30
list_learning_categories	31
list_name_regions	32
list_scenarios	32
list_simulators	33
plot.network_estimation	33
plot.tna_reliability_comparison	34
plot_network_estimation	35
plot_sampling_distribution	36
plot_tna_comparison	37
print.network_estimation	38
print.tna_reliability_comparison	39
run_bootstrap_iteration	39
run_bootstrap_simulation	42
run_grid_simulation	45
run_network_simulation	48
run_sampling_analysis	50
run_scenario	52
sample_tna	52
select_states	53
simulate	55
simulate_anova	56
simulate_clusters	57
simulate_correlation	58
simulate_data	59
simulate_edge_list	60
simulate_fa	62

simulate_group_tna_networks	63
simulate_growth	66
simulate_hmm	67
simulate_htna	69
simulate_igraph	71
simulate_irt	74
simulate_lca	76
simulate_long_data	77
simulate_longitudinal	80
simulate_lpa	82
simulate_matrix	83
simulate_mlm	85
simulate_network	86
simulate_onehot_data	89
simulate_prediction	92
simulate_regression	93
simulate_seq_clusters	94
simulate_sequences	95
simulate_sequences_advanced	99
simulate_survival	103
simulate_tna_datasets	105
simulate_tna_matrix	108
simulate_tna_network	111
simulate_tna_networks	112
simulate_ttest	114
summarize_grid_results	116
summarize_networks	118
summarize_simulation	120
tidy_simulation_results	121
validate_recovery	122
validate_sim_params	123

Index**125**

batch_apply

*Apply Function to Multiple Models or Datasets***Description**

Apply a function to a list of TNA models or datasets, with optional parallel processing.

Usage

```
batch_apply(
  object_list,
  fun,
  parallel = FALSE,
  cores = NULL,
```

```

    progress = TRUE,
    simplify = FALSE,
    ...
)

```

Arguments

<code>object_list</code>	A list of TNA model objects or data frames.
<code>fun</code>	A function to apply to each element.
<code>parallel</code>	Logical. Whether to use parallel processing. Default: FALSE.
<code>cores</code>	Integer or NULL. Number of cores for parallel processing. Default: NULL.
<code>progress</code>	Logical. Whether to show progress messages. Default: TRUE.
<code>simplify</code>	Logical. Whether to simplify results if possible. Default: FALSE.
<code>...</code>	Additional arguments passed to <code>fun</code> .

Details

This is a general-purpose function for batch operations on model lists. It handles errors gracefully, returning NULL for failed operations.

Value

A list of results from applying `fun` to each element. If `simplify = TRUE` and results are atomic, returns a simplified vector.

See Also

[batch_fit_models](#) for fitting multiple models.

Examples

```

## Not run:
# Extract transition matrices from multiple models
trans_mats <- batch_apply(models, extract_transition_matrix)

# Compare each model to a reference
ref_model <- models[[1]]
comparisons <- batch_apply(
  models[-1],
  function(m) compare_networks(ref_model, m)$metrics$correlation
)

# Extract centralities with simplification
centralities <- batch_apply(
  models,
  function(m) tna::centralities(m),
  simplify = FALSE
)

```

```
## End(Not run)
```

batch_fit_models	<i>Fit Models to Multiple Datasets</i>
------------------	--

Description

Fit TNA models to multiple datasets at once, with optional parallel processing.

Usage

```
batch_fit_models(
  data_list,
  model_type = c("tna", "ftna", "ctna", "atna"),
  parallel = FALSE,
  cores = NULL,
  progress = TRUE,
  ...
)
```

Arguments

<code>data_list</code>	A list of data frames, each containing sequence data.
<code>model_type</code>	Character. Type of model to fit: "tna", "ftna", "ctna", or "atna". Default: "tna".
<code>parallel</code>	Logical. Whether to use parallel processing. Default: FALSE.
<code>cores</code>	Integer or NULL. Number of cores to use for parallel processing. If NULL, uses <code>parallel::detectCores()</code> - 1. Ignored if <code>parallel = FALSE</code> . Default: NULL.
<code>progress</code>	Logical. Whether to show progress messages. Default: TRUE.
<code>...</code>	Additional arguments passed to the model fitting function.

Details

This function provides a convenient way to fit TNA models to many datasets, such as when running simulations or analyzing multiple groups.

For parallel processing on Windows, set up a parallel backend first using the `future` package.

Value

A list of fitted model objects, with the same names/indices as `data_list`. Failed fits are returned as NULL with a warning.

See Also

[fit_network_model](#) for fitting a single model, [batch_apply](#) for applying functions to model lists.

Examples

```
## Not run:
# Generate multiple datasets
datasets <- lapply(1:10, function(i) {
  simulate_sequences(trans_mat, init_probs, max_seq_length = 20, num_rows = 100)
})

# Fit models in sequence
models <- batch_fit_models(datasets, model_type = "tna")

# Fit models in parallel
models <- batch_fit_models(datasets, model_type = "tna",
  parallel = TRUE, cores = 4)

# Check results
sapply(models, function(m) if (!is.null(m)) "OK" else "Failed")

## End(Not run)
```

compare_centralities *Compare Centrality Profiles*

Description

Compare centrality measures between two TNA networks using correlation or rank-based methods.

Usage

```
compare_centralities(
  model1,
  model2,
  measures = c("OutStrength", "InStrength", "Betweenness"),
  method = c("both", "correlation", "rank")
)
```

Arguments

model1	A TNA model object (the reference/original model).
model2	A TNA model object (the comparison/simulated model).
measures	Character vector. Centrality measures to compare. Options: "OutStrength", "InStrength", "Betweenness", "Closeness", or "all". Default: c("OutStrength", "InStrength", "Betweenness").
method	Character. Comparison method: "correlation" (Pearson), "rank" (Spearman), or "both". Default: "both".

Details

Centrality measures are extracted from the TNA model objects using `tna::centralities()`. The function compares how well the centrality profiles are preserved between the original and simulated/comparison model.

Value

A list containing:

- `correlations`: Named list of correlation values by measure.
- `centrality_comparison`: Data frame with centrality values for each state.
- `summary`: Character summary of the comparison.

See Also

[compare_networks\(\)](#) for full network comparison.

Examples

```
## Not run:  
# Compare centrality profiles between models  
comparison <- compare_centralities(model_original, model_simulated)  
print(comparison$correlations)  
  
## End(Not run)
```

`compare_edge_recovery` *Calculate Edge Recovery Metrics*

Description

Calculate metrics for how well edges from an original network are recovered in a simulated/comparison network.

Usage

```
compare_edge_recovery(  
  original,  
  simulated,  
  threshold = 0.01,  
  return_edges = FALSE  
)  
  
calculate_edge_recovery(...)
```

Arguments

original	A TNA model object (the original/ground truth model).
simulated	A TNA model object (the simulated/recovered model).
threshold	Numeric. Minimum edge weight to consider an edge as "present". Default: 0.01.
return_edges	Logical. Whether to return detailed edge-level results. Default: FALSE.
...	Arguments passed to compare_edge_recovery.

Details

This function treats edge recovery as a binary classification problem:

- True Positive: Edge present in both original and simulated.
- False Positive: Edge present in simulated but not original.
- False Negative: Edge present in original but not simulated.
- True Negative: Edge absent in both.

An edge is considered "present" if its weight exceeds the threshold.

Value

A list containing:

- true_positives: Number of edges correctly present in both.
- false_positives: Number of edges present in simulated but not original.
- false_negatives: Number of edges present in original but not simulated.
- true_negatives: Number of edges correctly absent in both.
- precision: $TP / (TP + FP)$.
- recall: $TP / (TP + FN)$, also known as sensitivity.
- f1_score: Harmonic mean of precision and recall.
- accuracy: $(TP + TN) / \text{total edges}$.
- jaccard: $TP / (TP + FP + FN)$, Jaccard similarity.
- edges: (Optional) Data frame with edge-level results.

See Also

[compare_networks\(\)](#) for full network comparison, [run_bootstrap_iteration\(\)](#) for bootstrap evaluation.

Examples

```
## Not run:
# Calculate edge recovery
recovery <- compare_edge_recovery(model_original, model_simulated)
print(sprintf("Precision: %.2f, Recall: %.2f", recovery$precision, recovery$recall))

## End(Not run)
```

compare_estimation	<i>Compare Model Estimation Across Simulations</i>
--------------------	--

Description

Run multiple simulations comparing how well different TNA model types recover the true transition structure, using `tna::compare()` for each comparison. Supports any model type available in the `tna` package.

Usage

```
compare_estimation(
  models = c("tna", "ftna"),
  n_simulations = 1000,
  n_sequences = 200,
  seq_length = 25,
  n_states = 6,
  na_range = c(0, 5),
  scaling = "minmax",
  seed = NULL,
  verbose = TRUE,
  parallel = FALSE,
  cores = parallel::detectCores() - 1
)
```

Arguments

<code>models</code>	Character vector. Model types to compare. Any model function exported by <code>tna</code> package works: "tna", "ftna", "ctna", "atna", "sna", "tsn". Default: <code>c("tna", "ftna")</code> .
<code>n_simulations</code>	Integer. Number of simulations to run. Default: 1000.
<code>n_sequences</code>	Integer. Number of sequences per simulation. Default: 200.
<code>seq_length</code>	Integer. Maximum sequence length. Default: 25.
<code>n_states</code>	Integer. Number of states. Default: 6.
<code>na_range</code>	Integer vector of length 2. Range of NAs per sequence for varying lengths. Default: <code>c(0, 5)</code> .
<code>scaling</code>	Character. Scaling for <code>tna::compare()</code> . Default: "minmax".
<code>seed</code>	Integer or NULL. Random seed. Default: NULL.
<code>verbose</code>	Logical. Print progress. Default: TRUE.
<code>parallel</code>	Logical. Use parallel processing. Default: FALSE.
<code>cores</code>	Integer. Number of cores for parallel. Default: <code>detectCores() - 1</code> .

Details

For each simulation:

1. Generate random transition probabilities (ground truth)
2. Simulate sequences with optional NAs (varying lengths)
3. Fit all specified model types
4. Compare each to ground truth using `tna::compare()`
5. Collect metrics (Pearson, Spearman, Kendall, Euclidean, etc.)

Available model types (any `tna` package model):

- `tna`: Standard transition network analysis (probabilities)
- `ftna`: Frequency-based TNA (raw counts)
- `ctna`: Concurrent TNA
- `atna`: Absorbing TNA
- `sna`: Sequential network analysis
- `tsn`: Time-series network
- Any other model function exported by `tna` package

Value

A list containing:

comparison Side-by-side comparison of key metrics across models.

summary Data frame with mean/sd of all metrics by model type.

raw_results Data frame with all simulation results.

ranking Models ranked by Pearson correlation (best to worst).

winner Model with highest Pearson correlation.

params Parameters used for the simulation.

Examples

```
## Not run:
# Compare 2 models (default: tna vs ftna)
results <- compare_estimation(n_simulations = 100, seed = 42)

# Compare 3 models
results <- compare_estimation(
  models = c("tna", "ftna", "ctna"),
  n_simulations = 100,
  seed = 42
)

# Compare all 4 models
results <- compare_estimation(
  models = c("tna", "ftna", "ctna", "atna"),
```

```
n_simulations = 500,
parallel = TRUE,
seed = 42
)

# View results
results$comparison # Side-by-side metrics
results$ranking    # Best to worst
results$winner     # Top performer

## End(Not run)
```

`compare_network_estimation`*Compare Network Estimation Across TNA Model Types*

Description

Perform side-by-side comparison of multiple TNA model types (tna, atna, ctna, ftna) using comprehensive sampling analysis. Uses a statistically correct approach: comparing models built on independent data subsets (sample vs remaining).

Usage

```
compare_network_estimation(
  data,
  model_types = c("tna", "ftna"),
  model_scaling = NULL,
  sampling_percent = 0.3,
  iterations = 100,
  seed = NULL,
  verbose = TRUE
)

compare_tna_models(
  data,
  model_types = c("tna", "ftna"),
  model_scaling = NULL,
  sampling_percent = 0.3,
  iterations = 100,
  seed = NULL,
  verbose = TRUE
)
```

Arguments

<code>data</code>	A data frame containing sequence data.
<code>model_types</code>	Character vector of model types to compare. Valid types are "tna", "atna", "ctna", "ftna". Default: c("tna", "ftna").
<code>model_scaling</code>	A named list specifying scaling for each model type. Example: list(tna = "min-max", ftna = "skip"). If NULL, uses default scaling.
<code>sampling_percent</code>	Numeric value between 0 and 1 indicating the proportion of data to sample. Default: 0.3.
<code>iterations</code>	Integer specifying the number of sampling iterations for each model. Default: 100.
<code>seed</code>	Integer seed for reproducible random sampling. Default: NULL.
<code>verbose</code>	Logical. Print progress messages. Default: TRUE.

Details

The function workflow:

1. For each model type, builds a model from the data.
2. Runs sampling analysis comparing sample vs remaining subsets.
3. Combines all results for comparison.

This approach is statistically correct because it compares models built on independent data subsets, providing a true measure of estimation stability.

Value

A list containing:

individual Data frame with all iteration results including `model_type` column.

aggregated Data frame with aggregated metrics by model type.

ranking Models ranked by mean Pearson correlation.

winner Model with highest Pearson correlation.

params Parameters used for the comparison.

Examples

```
## Not run:
library(tna)
data(group_regulation)

# Compare tna and ftna models
results <- compare_network_estimation(group_regulation)
results$ranking
results$winner

# Compare all model types
```

```

results <- compare_network_estimation(
  group_regulation,
  model_types = c("tna", "atna", "ctna", "ftna"),
  iterations = 100
)

# Plot results
plot(results) # Default histogram
plot(results, metric_name = "Pearson")

## End(Not run)

```

compare_networks	<i>Compare Two TNA Networks</i>
------------------	---------------------------------

Description

Compare two TNA network models using multiple metrics including correlation, RMSE, and edge-level differences.

Usage

```

compare_networks(
  model1,
  model2,
  metrics = c("correlation", "rmse", "edge_diff"),
  scaling = c("none", "minmax", "zscore"),
  include_self = TRUE,
  threshold = 0.05
)

```

Arguments

model1	A TNA model object (the reference/original model).
model2	A TNA model object (the comparison/simulated model).
metrics	Character vector. Metrics to compute. Options: "correlation" (Pearson correlation), "rmse" (root mean square error), "mae" (mean absolute error), "edge_diff" (proportion with large differences), "cosine" (cosine similarity), or "all" to compute everything. Default: c("correlation", "rmse", "edge_diff").
scaling	Character. How to scale edge weights before comparison. Options: "none" (no scaling), "minmax" (min-max normalization to 0-1), "zscore" (z-score standardization). Default: "none".
include_self	Logical. Whether to include self-loops in comparison. Default: TRUE.
threshold	Numeric. Threshold for considering an edge as "different" in edge_diff metric. Default: 0.05.

Details

This function extracts edge weights from both models and computes various comparison metrics. Edge weights are extracted from the transition matrices stored in the TNA model objects.

Value

A list with class "tna_comparison" containing:

- metrics: Named list of computed metrics.
- edge_comparison: Data frame comparing edges from both models.
- summary: Character summary of the comparison.

See Also

[compare_centralities\(\)](#) for comparing centrality profiles, [compare_edge_recovery\(\)](#) for edge recovery metrics.

Examples

```
## Not run:
# Generate original data and fit model
original_data <- simulate_sequences(trans_mat, init_probs, 20, 200)
model_original <- tna::tna(original_data)

# Simulate from fitted model and fit new model
sim_data <- simulate_sequences(
  transition_matrix = model_original$weights,
  initial_probabilities = model_original$initial,
  max_seq_length = 20, num_rows = 200
)
model_sim <- tna::tna(sim_data)

# Compare the models
comparison <- compare_networks(model_original, model_sim)
print(comparison$metrics)

## End(Not run)
```

compare_reliability	<i>Compare Reliability Across Data Conditions</i>
---------------------	---

Description

Run multiple simulations assessing split-half reliability under varying data conditions (number of sequences, sequence length, number of states). Uses `tna::reliability()` for each simulation with randomly sampled parameters from user-specified ranges.

Usage

```

compare_reliability(
  n_simulations = 1000,
  n_sequences = c(50, 500),
  seq_length = c(10, 50),
  n_states = 6,
  alpha = 1,
  diag_c = 0,
  na_range = c(0, 5),
  model_type = "tna",
  reliability_iter = 100,
  reliability_split = 0.5,
  scaling = "none",
  seed = NULL,
  verbose = TRUE,
  parallel = FALSE,
  cores = parallel::detectCores() - 1
)

```

Arguments

<code>n_simulations</code>	Integer. Number of simulations to run. Default: 1000.
<code>n_sequences</code>	Integer or integer vector of length 2. Number of sequences per simulation. If length 2, each simulation samples a random integer in that range. Default: <code>c(50, 500)</code> .
<code>seq_length</code>	Integer or integer vector of length 2. Sequence length. If length 2, each simulation samples a random integer in that range. Default: <code>c(10, 50)</code> .
<code>n_states</code>	Integer or integer vector of length 2. Number of states. If length 2, each simulation samples a random integer in that range. Default: 6.
<code>alpha</code>	Numeric or numeric vector of length 2. Dirichlet concentration parameter for <code>generate_probabilities()</code> . If length 2, each simulation samples a random value from <code>runif(1, alpha[1], alpha[2])</code> . Small values (e.g., 0.1) produce sparse transition matrices; large values (e.g., 10) produce near-uniform matrices. Default: 1.
<code>diag_c</code>	Numeric or numeric vector of length 2. Diagonal boost for <code>generate_probabilities()</code> . If length 2, each simulation samples a random value from <code>runif(1, diag_c[1], diag_c[2])</code> . Higher values create "sticky" states with strong self-transitions. Default: 0.
<code>na_range</code>	Integer vector of length 2. Range of NAs per sequence. Default: <code>c(0, 5)</code> .
<code>model_type</code>	Character. Which model to fit. Default: "tna".
<code>reliability_iter</code>	Integer. Number of iterations for <code>tna::reliability()</code> . Default: 100.
<code>reliability_split</code>	Numeric. Split proportion for <code>tna::reliability()</code> . Default: 0.5.
<code>scaling</code>	Character. Scaling for <code>tna::reliability()</code> . Default: "none".

seed	Integer or NULL. Random seed. Default: NULL.
verbose	Logical. Print progress. Default: TRUE.
parallel	Logical. Use parallel processing. Default: FALSE.
cores	Integer. Number of cores for parallel. Default: detectCores() - 1.

Details

For each simulation:

1. Randomly sample `n_sequences`, `seq_length`, `n_states` from user-specified ranges.
2. Generate ground truth via `generate_probabilities()`.
3. Simulate sequences via `simulate_sequences()` with those parameters and NAs.
4. Fit a TNA model (e.g., `tna::tna(sequences)`).
5. Run `tna::reliability()` on the fitted model.
6. Extract the `$summary` table (22 metrics with mean/sd/median/min/max/q25/q75).
7. Append simulation parameters as columns.

Value

A list of class "tna_reliability_comparison" containing:

raw_results Data frame with all simulation results (one row per metric per simulation, with parameter columns).

summary Data frame with mean/sd aggregated across simulations grouped by metric.

params List of input parameters for reference.

Examples

```
## Not run:
# Quick test
res <- compare_reliability(
  n_simulations = 10,
  reliability_iter = 20,
  seed = 42
)
print(res)

# Vary number of states too
res <- compare_reliability(
  n_simulations = 100,
  n_states = c(3, 8),
  reliability_iter = 50,
  seed = 42
)
plot(res)

# Parallel execution
res <- compare_reliability(
```

```
n_simulations = 500,  
parallel = TRUE,  
seed = 42  
)  
  
## End(Not run)
```

cross_validate_tna	<i>Cross-Validate TNA Model Types</i>
--------------------	---------------------------------------

Description

Perform cross-validation by testing different TNA model types on the same data, allowing comparison of model performance.

Usage

```
cross_validate_tna(  
  data,  
  model_types = c("relative", "frequency", "co-occurrence"),  
  sampling_percent = 0.3,  
  iterations = 50,  
  seed = NULL,  
  verbose = TRUE  
)
```

Arguments

data	A data frame containing sequence data.
model_types	Character vector of model types to test. Default: c("relative", "frequency", "co-occurrence").
sampling_percent	Proportion of data for sampling. Default: 0.3.
iterations	Number of iterations per model type. Default: 50.
seed	Random seed for reproducibility. Default: NULL.
verbose	Logical. Print progress. Default: TRUE.

Value

A list containing cross-validation results for each model type.

Examples

```
## Not run:
data(group_regulation, package = "tna")
cv_results <- cross_validate_tna(
  group_regulation,
  model_types = c("relative", "frequency"),
  iterations = 30
)

## End(Not run)
```

export_simulation	<i>Export Simulation Results to CSV</i>
-------------------	---

Description

Write a tidy simulation data.frame to CSV. A list of [saqr_sim](#) objects (or a single one) is first flattened with [tidy_simulation_results](#); a data.frame is written as-is.

Usage

```
export_simulation(x, file)
```

Arguments

x	A data.frame, a single saqr_sim object, or a list of saqr_sim objects.
file	Character scalar. Path of the CSV file to write.

Value

The file path, invisibly.

Examples

```
s <- run_scenario("power_ttest", seed = 1)
path <- tempfile(fileext = ".csv")
export_simulation(s, path)
```

fit_network_model	<i>Fit a Temporal Network Analysis Model</i>
-------------------	--

Description

Fit a TNA (Temporal Network Analysis) model to sequence data using one of several available model types.

Usage

```
fit_network_model(sequences, model_type, group = NULL)
```

Arguments

sequences	Data frame of sequences. Each row is a sequence, each column is a time point. For group models, should include a grouping column.
model_type	Character. Type of model to fit. One of: "tna" Standard Temporal Network Analysis model. "ftna" Filtered TNA model. "ctna" Conditional TNA model. "atna" Aggregated TNA model. "group_tna" Group TNA model for multi-group analysis.
group	Character or NULL. Name of the grouping column for group_tna models. Required when model_type = "group_tna". Default: NULL.

Details

This function is a wrapper around the tna package's model fitting functions. It provides a unified interface for fitting different types of TNA models.

For group_tna models, the sequences data frame must contain a column identifying the group membership of each sequence, specified by the group parameter.

Value

A fitted TNA model object of the appropriate class.

Examples

```
## Not run:
# Generate some sequence data
trans_mat <- matrix(c(
  0.7, 0.2, 0.1,
  0.3, 0.5, 0.2,
  0.2, 0.3, 0.5
), nrow = 3, byrow = TRUE)
rownames(trans_mat) <- colnames(trans_mat) <- c("A", "B", "C")
```

```

init_probs <- c(A = 0.5, B = 0.3, C = 0.2)

sequences <- simulate_sequences(
  transition_matrix = trans_mat,
  initial_probabilities = init_probs,
  max_seq_length = 20,
  num_rows = 100
)

# Fit different model types
model_tna <- fit_network_model(sequences, "tna")
model_ftna <- fit_network_model(sequences, "ftna")

# Fit group model
long_data <- simulate_long_data(n_groups = 5, actors_per_group = 10)
wide_data <- long_to_wide(long_data, id_col = "Actor")
# Add group column back
actor_groups <- unique(long_data[, c("Actor", "Group")])
wide_data <- merge(wide_data, actor_groups, by = "Actor")
model_group <- fit_network_model(wide_data, "group_tna", group = "Group")

## End(Not run)

```

generate_param_grid *Generate Parameter Grid for Simulations*

Description

Generate a grid of parameter combinations for simulation studies using various sampling methods. Can be called with no arguments for a demo grid.

Usage

```

generate_param_grid(param_ranges = NULL, n = 10, method = "random")

create_param_grid(...)

```

Arguments

param_ranges	Named list of parameter ranges. Each element can be: • A numeric vector of length 2 (min, max) for continuous/integer parameters. • A vector of values for categorical parameters. If NULL (default), uses demo ranges for TNA simulation: <code>list(n_sequences = c(50, 500), seq_length = c(10, 50), n_states = c(4, 12))</code> .
n	Integer. Number of parameter combinations to generate. Default: 10.
method	Character. Sampling method. One of:

"**random**" Random uniform sampling within ranges.
 "**grid**" Regular grid sampling (may exceed n, then subsampled).
 "**lhs**" Latin Hypercube Sampling for better coverage (requires lhs package).
 Default: "random".
 ... Arguments passed to generate_param_grid.

Details

Method Details:

- "random": Draws uniform random values within each range. Integer parameters (detected when min and max are both integers) are rounded.
- "grid": Creates a regular grid with approximately $n^{1/d}$ points per dimension (where d is the number of parameters). If the resulting grid exceeds n points, it is randomly subsampled.
- "lhs": Uses Latin Hypercube Sampling for space-filling designs that provide better coverage of the parameter space than random sampling.

Categorical parameters are sampled uniformly with replacement for all methods.

Value

A data frame with n rows and columns for each parameter.

Examples

```
# Simplest usage: demo grid with default TNA parameters
grid <- generate_param_grid()
head(grid)

# Define custom parameter ranges
ranges <- list(
  num_rows = c(50, 500),      # Integer parameter
  max_seq_length = c(10, 100), # Integer parameter
  stability_prob = c(0.7, 1.0) # Continuous parameter
)

# Random sampling
grid_random <- generate_param_grid(ranges, n = 20, method = "random")

# Latin Hypercube Sampling
grid_lhs <- generate_param_grid(ranges, n = 20, method = "lhs")

# Grid sampling
grid_regular <- generate_param_grid(ranges, n = 20, method = "grid")
```

generate_probabilities

Generate Transition and Initial Probabilities

Description

Generate random transition probabilities and initial state probabilities for a Markov chain using the seqHMM package.

Usage

```
generate_probabilities(
  n_states = 8,
  states = NULL,
  alpha = 1,
  diag_c = 0,
  seed = NULL,
  possible_state_names = NULL
)
```

Arguments

n_states	Integer. Number of states in the Markov chain. Default: 5.
states	Character vector. Names for the states. Must have at least n_states elements. If NULL, uses letters A, B, C, ... Default: NULL.
alpha	Numeric. Dirichlet concentration parameter passed to seqHMM::simulate_initial_probs() and seqHMM::simulate_transition_probs() . Small values (e.g., 0.1) produce sparse matrices with a few dominant transitions; large values (e.g., 10) produce near-uniform matrices. Default: 1.
diag_c	Numeric. Diagonal boost added before row normalisation, passed to seqHMM::simulate_transition_probs() . Higher values create "sticky" states with strong self-transitions. Default: 0.
seed	Integer or NULL. Random seed for reproducibility. Default: NULL.
possible_state_names	Deprecated. Use states instead.

Details

The transition probabilities are generated using [seqHMM::simulate_transition_probs\(\)](#) and initial probabilities using [seqHMM::simulate_initial_probs\(\)](#).

Value

A list containing:

initial_probs Named numeric vector of initial state probabilities.

transition_probs Square matrix of transition probabilities with row and column names set to state names.

state_names Character vector of state names used.

Examples

```
## Not run:
# Generate probabilities for 4 states with default names
probs <- generate_probabilities(n_states = 4, seed = 42)

# Generate probabilities with custom state names
probs <- generate_probabilities(
  n_states = 4,
  states = c("A", "B", "C", "D", "E"),
  seed = 123
)

# View initial probabilities
probs$initial_probs

# View transition matrix
probs$transition_probs

# Old parameter name still works
probs <- generate_probabilities(
  n_states = 3,
  possible_state_names = c("X", "Y", "Z")
)

## End(Not run)
```

get_global_names

*Get Global Names***Description**

Retrieve names from the global names dataset, with optional filtering by region.

Usage

```
get_global_names(n = NULL, regions = "all", seed = NULL)
```

Arguments

n	Integer or NULL. Number of names to return. If NULL, returns all matching names. Default: NULL.
regions	Character vector. Regions to sample from. Can be: • "all": All regions (default) • Specific regions: e.g., "arab", "east_asia", "nordic" • Shortcuts: e.g., "europe", "africa", "asia" • Multiple: e.g., c("arab", "persian", "turkish") Default: "all".
seed	Integer or NULL. Random seed for reproducibility. Default: NULL.

Value

Character vector of names.

See Also

[list_name_regions](#) for available regions, [GLOBAL_NAMES](#) for the full dataset.

Examples

```
# Get 20 random names from all regions
get_global_names(20, seed = 42)

# Get 10 names from Arab region
get_global_names(10, regions = "arab")

# Get 15 names from any African region
get_global_names(15, regions = "africa")

# Get names from multiple regions
get_global_names(20, regions = c("east_asia", "south_asia"))

# Get all Nordic names
get_global_names(regions = "nordic")
```

get_learning_states	<i>Get Learning State Verbs</i>
---------------------	---------------------------------

Description

Retrieve learning action verbs by category, with options for random selection and combining multiple categories.

Usage

```
get_learning_states(categories = "all", n = NULL, seed = NULL)
```

Arguments

categories	Character vector specifying which categories to include. Options: "metacognitive", "cognitive", "behavioral", "social", "motivational", "affective", or "all" for all categories. Default: "all".
n	Integer or NULL. If specified, randomly sample n verbs from the selected categories. If NULL, return all verbs. Default: NULL.
seed	Integer or NULL. Random seed for reproducible sampling. Default: NULL.

Value

Character vector of learning state verbs.

Examples

```
# All metacognitive verbs
get_learning_states("metacognitive")

# 10 random verbs from all categories
get_learning_states(n = 10, seed = 42)

# 5 verbs from cognitive and behavioral
get_learning_states(c("cognitive", "behavioral"), n = 5)

# Self-regulated learning focus (metacognitive + motivational)
get_learning_states(c("metacognitive", "motivational"))
```

get_scenario

Get a Scenario Recipe

Description

Retrieve the argument-lists for a named scenario. With `case = NULL` the full list of argument-lists is returned; with an integer `case` the single argument-list for that case is returned, ready to `do.call()` against the scenario's simulator.

Usage

```
get_scenario(scenario, case = NULL)
```

Arguments

scenario	Character scalar. The scenario name (see list_scenarios).
case	Integer scalar or NULL. When NULL (default) the full list of argument-lists is returned. When an integer, the single argument-list at that index is returned.

Value

When `case = NULL`, a list of named argument-lists. When `case` is an integer, a single named argument-list.

Examples

```
get_scenario("power_ttest")
get_scenario("power_ttest", case = 1)
```

GLOBAL_NAMES

*Global Names Dataset***Description**

A collection of ~1000 culturally diverse names from around the world, organized by region for use in social network simulations.

Usage

GLOBAL_NAMES

Format

A named list of character vectors, with each element containing names from a specific region.

Details

Regions are organized geographically and include:

Europe:

- western_europe: UK, Ireland, France, Germany, Netherlands, Belgium
- eastern_europe: Russia, Ukraine, Poland, Czech, Romania, Hungary, Bulgaria
- southern_europe: Spain, Italy, Portugal, Greece
- nordic: Sweden, Norway, Denmark, Finland, Iceland

Middle East & Central Asia:

- arab: Gulf states, Levant, Egypt
- persian: Iran, Afghanistan, Tajikistan
- turkish: Turkey, Azerbaijan
- central_asia: Kazakhstan, Uzbekistan, Kyrgyzstan, Turkmenistan

Asia:

- south_asia: India, Pakistan, Bangladesh, Sri Lanka, Nepal
- east_asia: China, Japan, Korea, Mongolia
- southeast_asia: Vietnam, Thailand, Indonesia, Philippines, Malaysia

Africa:

- west_africa: Nigeria, Ghana, Senegal, Mali
- east_africa: Kenya, Ethiopia, Tanzania, Somalia, Uganda
- southern_africa: South Africa, Zimbabwe, Botswana, Namibia
- north_africa: Morocco, Algeria, Tunisia, Libya
- central_africa: DRC, Cameroon, Congo

Americas:

- north_america: USA, Canada
- latin_america: Mexico, Central America, South America, Brazil
- caribbean: Jamaica, Haiti, Cuba, Trinidad, Dominican Republic
- indigenous_americas: Native American, First Nations, Mayan, Incan

Oceania:

- australia_nz: Australia, New Zealand
- pacific_islands: Hawaii, Samoa, Tonga, Fiji, PNG
- indigenous_oceania: Maori, Aboriginal Australian

Other:

- jewish_hebrew: Hebrew/Jewish names
- armenian_georgian: Armenia, Georgia

See Also

[get_global_names](#) for sampling names, [list_name_regions](#) for exploring available regions.

Examples

```
# View all region names
names(GLOBAL_NAMES)

# View names from a specific region
GLOBAL_NAMES$arab
GLOBAL_NAMES$east_asia

# Count names per region
sapply(GLOBAL_NAMES, length)

# Total names available
sum(sapply(GLOBAL_NAMES, length))
```

GROUP_REGULATION_ACTIONS

Group Regulation Actions

Description

The 9 standard group regulation (SSRL) actions used in collaborative learning research.

Usage

GROUP_REGULATION_ACTIONS

Format

A character vector of 9 actions.

Details

These actions represent Socially Shared Regulation of Learning (SSRL):

- adapt** Adjusting strategies based on feedback
- cohesion** Building and maintaining group unity
- consensus** Reaching group agreement
- coregulate** Mutually regulating each other's learning
- discuss** Engaging in group discussion
- emotion** Expressing or regulating emotions
- monitor** Tracking and evaluating progress
- plan** Planning group activities and strategies
- synthesis** Integrating and combining ideas

Examples

```
GROUP_REGULATION_ACTIONS

# Use in simulation
data <- simulate_long_data(
  n_groups = 10,
  states = GROUP_REGULATION_ACTIONS,
  seed = 42
)
```

inject_missingness	<i>Inject Missing Values Under a Known Mechanism (MCAR / MAR / MNAR)</i>
--------------------	--

Description

Take an existing complete data.frame and inject missing values (NA) into target columns under a chosen, well-defined missingness mechanism. This is a *transformer*, not a generator: the returned object is the same data.frame with some cells replaced by NA, carrying an attached "missing_info" attribute that records exactly which cells were made missing and how. Useful for testing imputation methods and missing-data-aware estimators against a known ground truth.

The three mechanisms follow the standard Rubin taxonomy:

"MCAR" Missing Completely At Random. Each cell of the target columns is independently set missing with probability prop.

"MAR" Missing At Random. The probability that a target cell is missing depends on an *observed* predictor column, never on the to-be-missing value itself. Probabilities are calibrated (via a rank/logistic weighting on the predictor) so the average missing fraction is approximately prop.

"MNAR" Missing Not At Random. The probability that a target cell is missing depends on that cell's *own* (numeric) value: larger values are more likely to go missing. Calibrated so the average missing fraction per column is approximately prop.

Usage

```
inject_missingness(
  data,
  mechanism = c("MCAR", "MAR", "MNAR"),
  prop = 0.1,
  cols = NULL,
  predictor = NULL,
  seed = NULL
)
```

Arguments

data	A data.frame, or a saqr_sim object (its \$data is used). The complete data into which to inject missingness.
mechanism	Character. One of "MCAR", "MAR", "MNAR" (matched via match.arg).
prop	Numeric scalar in [0, 1]. Target average fraction of cells in the target columns to set missing. Default: 0.1.
cols	Character vector of target column names, or NULL (the default) meaning all columns. For "MAR", the predictor column is automatically excluded from the targets.
predictor	Character scalar naming the observed column that drives missingness under "MAR". Required for "MAR", ignored otherwise.
seed	Integer or NULL. If non-NULL, <code>set.seed()</code> is called for reproducibility.

Value

The input data.frame with NAs injected into the target columns. The returned object additionally carries an attribute "missing_info", a list with elements:

mechanism	The resolved mechanism string.
prop	The requested target proportion.
cols	The target column names actually used.
n_missing	Total number of cells set missing.
realized_prop	Realized fraction of target cells set missing ($n_missing / (nrow * length(cols))$).
indicator	A logical matrix ($nrow(data) \times length(cols)$) marking the cells that were injected as TRUE, with column names matching cols.

Examples

```
df <- simulate_prediction(
  n = 500,
  coefs = c("(Intercept)" = 1, x1 = 2, x2 = -1),
  seed = 1
)$data

# MCAR: 20% of every column missing at random
mcar <- inject_missingness(df, mechanism = "MCAR", prop = 0.2, seed = 7)
attr(mcar, "missing_info")$realized_prop
str(attr(mcar, "missing_info"))

# MAR: missingness in y driven by observed x1
mar <- inject_missingness(df, mechanism = "MAR", prop = 0.2,
  cols = "y", predictor = "x1", seed = 7)
colSums(is.na(mar))

# MNAR: large y-values more likely missing
mnar <- inject_missingness(df, mechanism = "MNAR", prop = 0.2,
  cols = "y", seed = 7)
attr(mnar, "missing_info")$n_missing
```

learning_states

Learning State Verbs for TNA Simulation

Description

Predefined collections of student learning action verbs organized by category. These can be used as state names when generating TNA networks to simulate realistic student learning behavior sequences.

Usage

```
LEARNING_STATES
```

Format

A named list with 8 categories of learning verbs.

Details

The verbs are organized into seven categories based on learning science:

metacognitive Self-regulation and awareness actions: planning, monitoring, evaluating one's own learning.

cognitive Mental processing actions: understanding, analyzing, synthesizing information.

behavioral Observable study actions: reading, writing, practicing, note-taking.

social Interaction and collaboration actions: discussing, helping, teaching, asking.
motivational Effort and persistence actions: focusing, persisting, engaging, striving.
affective Emotional and attitudinal states: enjoying, coping, managing stress, curiosity.
group_regulation Socially shared regulation of learning (SSRL): collaborative planning, monitoring, adapting in groups.
lms Learning Management System actions: viewing content, accessing resources, submitting assignments, forum posts.

Examples

```
# Get all available categories
names(get_learning_states())

# Get cognitive verbs only
get_learning_states("cognitive")

# Get multiple categories
get_learning_states(c("metacognitive", "cognitive"))

# Get all verbs combined
get_learning_states("all")

# Random selection of 8 verbs
get_learning_states(n = 8)

# Random 6 verbs from social and cognitive
get_learning_states(c("social", "cognitive"), n = 6)
```

```
list_learning_categories
```

Get Learning State Categories Summary

Description

Display a summary of all available learning state categories and the number of verbs in each.

Usage

```
list_learning_categories()
```

Value

A data frame with category names and verb counts.

Examples

```
list_learning_categories()
```

list_name_regions	<i>List Available Name Regions</i>
-------------------	------------------------------------

Description

Display all available regions and shortcuts for selecting names from GLOBAL_NAMES.

Usage

```
list_name_regions()
```

Value

A data frame with region names, counts, and example names.

Examples

```
list_name_regions()
```

list_scenarios	<i>List Available Scenario Presets</i>
----------------	--

Description

Return a tidy catalogue of the built-in scenario presets. Each scenario is a hard-coded recipe of argument-lists targeting one `simulate_*` function. Print the result directly.

Usage

```
list_scenarios()
```

Value

A base `data.frame` with one row per scenario and columns `scenario`, `description`, `simulator` (the target `simulate_*` function) and `n_cases` (number of argument-lists in the recipe).

Examples

```
list_scenarios()
```

list_simulators	<i>List Available Simulators</i>
-----------------	----------------------------------

Description

Return the catalogue of simulation types reachable through [simulate](#), one row per type, with the underlying function name and a one-line description. Print it directly.

Note: this lists the explicit-parameter simulators (those returning a [saqr_sim](#)). The separate [simulate_data](#) entry point provides random, seed-driven generation and returns a bare `data.frame`; it is not part of this catalogue.

Usage

```
list_simulators()
```

Value

A base `data.frame` with one row per simulator and columns:

`type` The type string passed to [simulate](#).

`function` The name of the underlying simulator function.

`description` A short one-line description.

See Also

[simulate](#)

Examples

```
list_simulators()
```

plot.network_estimation

Plot Network Estimation Comparison Results

Description

Plot method for `network_estimation` objects. Default is histogram with density overlay showing mean and median values.

Usage

```
## S3 method for class 'network_estimation'
plot(x, metric_name = "Pearson", plot_type = "histogram", ...)
```

Arguments

x	A network_estimation object from compare_network_estimation().
metric_name	Character. Metric to plot. Default: "Pearson".
plot_type	Character. Plot type: "histogram" (default), "boxplot", "density", "bar", or "ridgeline".
...	Additional arguments (currently unused).

Value

A ggplot object.

Examples

```
## Not run:
results <- compare_network_estimation(group_regulation)
plot(results) # Histogram of Pearson correlation
plot(results, metric_name = "Spearman")
plot(results, plot_type = "boxplot")

## End(Not run)
```

plot.tna_reliability_comparison

Plot Method for TNA Reliability Comparison

Description

Plot Method for TNA Reliability Comparison

Usage

```
## S3 method for class 'tna_reliability_comparison'
plot(x, metric = "Pearson", type = "scatter", ...)
```

Arguments

x	A tna_reliability_comparison object.
metric	Character. Metric to plot. Default: "Pearson".
type	Character. Plot type: "scatter", "boxplot", or "histogram". Default: "scatter".
...	Additional arguments (ignored).

`plot_network_estimation`*Plot Network Estimation Results*

Description

Create publication-ready plots for network estimation comparison results. Default is a histogram optimized for high iteration counts.

Usage

```
plot_network_estimation(  
  results,  
  metric_name = "Pearson",  
  plot_type = "histogram",  
  show_stats = TRUE,  
  bins = NULL  
)
```

Arguments

<code>results</code>	A <code>network_estimation</code> object or data frame with individual results.
<code>metric_name</code>	Character. Metric to plot. Default: "Pearson".
<code>plot_type</code>	Character. One of "histogram" (default), "boxplot", "density", "bar", or "ridge-line".
<code>show_stats</code>	Logical. Show mean/median statistics. Default: TRUE.
<code>bins</code>	Integer. Number of bins for histogram. Default: auto-calculated based on iterations (uses Sturges' rule).

Details

The histogram plot is optimized for high iteration counts (100+) and shows:

- Turquoise bars with density overlay
- Green vertical line for mean
- Orange dotted line for median
- Subtitle with exact mean and median values

Value

A ggplot object.

Examples

```
## Not run:
results <- compare_network_estimation(group_regulation, iterations = 100)

# Default histogram
plot_network_estimation(results)

# Different metrics
plot_network_estimation(results, metric_name = "Spearman")
plot_network_estimation(results, metric_name = "Euclidean")

# Different plot types
plot_network_estimation(results, plot_type = "boxplot")
plot_network_estimation(results, plot_type = "density")

## End(Not run)
```

plot_sampling_distribution

Plot Sampling Distribution for a Single Metric

Description

Create a visualization showing the distribution of a single metric's values across all sampling iterations.

Usage

```
plot_sampling_distribution(individual_results, category, metric)
```

Arguments

individual_results	Data frame of raw metric values from iterations, as returned by run_sampling_analysis() or compare_tna_models().
category	The category of the metric to plot (e.g., "Correlations").
metric	The specific metric name to plot (e.g., "Pearson").

Value

A ggplot object showing the distribution.

Examples

```
## Not run:
library(tna)
model <- tna(group_regulation)
results <- run_sampling_analysis(model, iterations = 100)

# Plot Pearson correlation distribution
plot_sampling_distribution(
  results$individual,
  category = "Correlations",
  metric = "Pearson"
)

## End(Not run)
```

plot_tna_comparison	<i>Plot TNA Model Comparison Results</i>
---------------------	--

Description

Visualize the distribution of metrics from TNA model comparisons. Supports multiple plot types including histogram (default), boxplot, ridgeline, bar, and density plots.

Usage

```
plot_tna_comparison(
  results,
  plot_type = "histogram",
  metric_name = "Pearson",
  model_types = NULL,
  binwidth = NULL
)
```

Arguments

results	A data frame containing comparison results from <code>compare_network_estimation()</code> (the individual element) or <code>run_sampling_analysis()</code> .
plot_type	Character string specifying the plot type: "histogram" (default), "boxplot", "ridgeline", "bar", or "density".
metric_name	Character string specifying a single metric to plot. Default: "Pearson". If NULL, all metrics will be plotted.
model_types	Character vector specifying the models to include. If NULL (default), all models in the data will be used.
binwidth	Numeric value for histogram bin width. If NULL (default), calculated automatically.

Details

Available plot types:

boxplot Box plots showing distribution by model type.

ridgeline Ridge line plots for density comparison.

bar Bar plots with mean values and error bars.

histogram Histogram with density overlay (requires metric_name).

density Density plots with mean/median lines (requires metric_name).

Value

A ggplot object.

Examples

```
## Not run:
library(tna)
data(group_regulation)

# Run comparison
results <- compare_tna_models(
  group_regulation,
  model_types = c("tna", "atna"),
  iterations = 30
)

# Boxplot of Pearson correlation
plot_tna_comparison(results$individual, "boxplot", metric_name = "Pearson")

# Histogram of single metric
plot_tna_comparison(results$individual, "histogram", metric_name = "Pearson")

# All metrics as bar chart
plot_tna_comparison(results$individual, "bar")

## End(Not run)
```

```
print.network_estimation
```

Print Network Estimation Results

Description

Print Network Estimation Results

Usage

```
## S3 method for class 'network_estimation'
print(x, ...)
```

Arguments

x	A network_estimation object.
...	Additional arguments (unused).

```
print.tna_reliability_comparison
```

Print Method for TNA Reliability Comparison

Description

Print Method for TNA Reliability Comparison

Usage

```
## S3 method for class 'tna_reliability_comparison'
print(x, ...)
```

Arguments

x	A tna_reliability_comparison object.
...	Additional arguments (ignored).

```
run_bootstrap_iteration
```

Run Bootstrap Iteration for a Single Simulation Run

Description

Performs a single bootstrap evaluation run: generates sequences, fits a TNA model, runs bootstrap analysis, and computes classification metrics for edge detection compared to ground truth stable transitions.

Usage

```
run_bootstrap_iteration(
  trans_matrix = NULL,
  init_probs = NULL,
  stable_transitions,
  seq_length = 20,
  n_sequences = 100,
  stability_prob = 0.95,
  unstable_mode = "random_jump",
  unstable_random_transition_prob = 0.5,
  unstable_perturb_noise = 0.5,
  unlikely_prob_threshold = 0.1,
  na_range = c(0, 0),
  include_na = TRUE,
  consistency_range = c(0.75, 1.25),
  level = 0.05,
  transition_matrix = NULL,
  initial_probabilities = NULL,
  max_seq_length = NULL,
  num_rows = NULL,
  min_na = NULL,
  max_na = NULL
)

evaluate_bootstrap(...)
```

Arguments

trans_matrix	Square numeric matrix of transition probabilities. Rows must sum to 1. Row names define state names.
init_probs	Named numeric vector of initial state probabilities. Must sum to 1.
stable_transitions	List of character vectors. Each vector contains two state names defining a stable (ground truth) transition pair.
seq_length	Integer. Maximum length of each sequence.
n_sequences	Integer. Number of sequences to generate.
stability_prob	Numeric in (0 to 1). Probability of following stable transitions.
unstable_mode	Character. Mode for unstable transitions: "random_jump", "perturb_prob", or "unlikely_jump".
unstable_random_transition_prob	Numeric in (0 to 1). Probability of unstable action.
unstable_perturb_noise	Numeric in (0 to 1). Noise factor for perturbation mode.
unlikely_prob_threshold	Numeric in (0 to 1). Threshold for unlikely transitions.

<code>na_range</code>	Integer vector of length 2 (min, max) or single integer (min=max). Range of NA values per sequence. Default: <code>c(0, 0)</code> .
<code>include_na</code>	Logical. Whether to include NAs.
<code>consistency_range</code>	Numeric vector of length 2. Range for bootstrap consistency analysis (e.g., <code>c(0.75, 1.25)</code>).
<code>level</code>	Numeric in (0,1). Significance level for p-value threshold (e.g., 0.05).
<code>transition_matrix</code>	Deprecated. Use <code>trans_matrix</code> instead.
<code>initial_probabilities</code>	Deprecated. Use <code>init_probs</code> instead.
<code>max_seq_length</code>	Deprecated. Use <code>seq_length</code> instead.
<code>num_rows</code>	Deprecated. Use <code>n_sequences</code> instead.
<code>min_na</code>	Deprecated. Use <code>na_range</code> instead.
<code>max_na</code>	Deprecated. Use <code>na_range</code> instead.
<code>...</code>	Arguments passed to <code>run_bootstrap_iteration</code> .

Details

The function:

1. Generates sequences using `simulate_sequences_advanced()`.
2. Fits a TNA model and runs bootstrap analysis.
3. Creates a ground truth matrix from `stable_transitions`.
4. Compares bootstrap-significant edges to ground truth.
5. Computes TP, TN, FP, FN and derived metrics per edge.

An edge is considered "significant" if its p-value < level.

Value

A list containing:

per_edge Data frame with per-edge results including ground truth, bootstrap significance, TP/TN/FP/FN, p-values, and metrics.

bootstrap_summary_raw Raw bootstrap summary from `tna::bootstrap`.

TP_matrix Matrix of true positives by edge.

TN_matrix Matrix of true negatives by edge.

FP_matrix Matrix of false positives by edge.

FN_matrix Matrix of false negatives by edge.

Returns NULL if bootstrap fails.

Examples

```
## Not run:
# Create transition matrix
trans_mat <- matrix(c(
  0.6, 0.3, 0.1,
  0.2, 0.6, 0.2,
  0.1, 0.2, 0.7
), nrow = 3, byrow = TRUE)
rownames(trans_mat) <- colnames(trans_mat) <- c("A", "B", "C")
init_probs <- c(A = 0.33, B = 0.34, C = 0.33)

# Define ground truth stable edges
stable <- list(c("A", "B"), c("B", "C"))

# Run single bootstrap evaluation
result <- run_bootstrap_iteration(
  trans_matrix = trans_mat,
  init_probs = init_probs,
  stable_transitions = stable,
  seq_length = 30,
  n_sequences = 100,
  stability_prob = 0.95,
  unstable_mode = "random_jump",
  unstable_random_transition_prob = 0.5,
  na_range = c(0, 5),
  include_na = TRUE,
  consistency_range = c(0.75, 1.25),
  level = 0.05
)

# Old parameter names still work
result <- run_bootstrap_iteration(
  transition_matrix = trans_mat,
  initial_probabilities = init_probs,
  stable_transitions = stable,
  max_seq_length = 30,
  num_rows = 100
)

## End(Not run)
```

run_bootstrap_simulation

Run Bootstrap Simulation Across Multiple Runs

Description

Simulate and analyze edge recovery across multiple bootstrap runs. Aggregates results to compute overall performance metrics and edge-level significance.

Usage

```
run_bootstrap_simulation(
  Model,
  stable_transitions,
  num_runs,
  seq_length = 20,
  n_sequences = 100,
  stability_prob = 0.95,
  unstable_mode = "random_jump",
  unstable_random_transition_prob = 0.5,
  unstable_perturb_noise = 0.5,
  unlikely_prob_threshold = 0.1,
  na_range = c(0, 0),
  include_na = TRUE,
  consistency_range = c(0.75, 1.25),
  level = 0.05,
  num_cores = parallel::detectCores() - 1,
  max_seq_length = NULL,
  num_rows = NULL,
  min_na = NULL,
  max_na = NULL
)
```

Arguments

Model	A TNA model object with weights (transition matrix) and inits (initial probabilities).
stable_transitions	List of character vectors defining ground truth stable transitions. Each vector contains two state names (from, to).
num_runs	Integer. Number of simulation runs to perform.
seq_length	Integer. Maximum sequence length.
n_sequences	Integer. Number of sequences per run.
stability_prob	Numeric in (0 to 1). Probability of following stable transitions. Default: 0.95.
unstable_mode	Character. Mode for unstable transitions: "random_jump", "perturb_prob", or "unlikely_jump". Default: "random_jump".
unstable_random_transition_prob	Numeric in (0 to 1). Probability of unstable action. Default: 0.5.
unstable_perturb_noise	Numeric in (0 to 1). Noise for perturbation mode. Default: 0.5.
unlikely_prob_threshold	Numeric in (0 to 1). Threshold for unlikely transitions. Default: 0.1.
na_range	Integer vector of length 2 (min, max) or single integer. Range of NA values per sequence. Default: c(0, 0).
include_na	Logical. Whether to include NAs. Default: TRUE.

consistency_range	Numeric vector of length 2. Bootstrap consistency range. Default: c(0.75, 1.25).
level	Numeric in (0,1). Significance level. Default: 0.05.
num_cores	Integer. Number of cores for parallel processing. Default: detectCores() - 1.
max_seq_length	Deprecated. Use seq_length instead.
num_rows	Deprecated. Use n_sequences instead.
min_na	Deprecated. Use na_range instead.
max_na	Deprecated. Use na_range instead.

Details

For each run, the function calls `run_bootstrap_iteration()` and accumulates TP/TN/FP/FN counts. After all runs, it computes:

- **Overall Performance:** Aggregated sensitivity (TPR), specificity (TNR), false positive rate (FPR), and false negative rate (FNR).
- **Edge Significance:** For each edge, the number of times it was detected as significant, average p-value, average weight, and recovery rate.

Parallel processing uses `parallel::mclapply()`.

Value

A list containing:

aggregated_summary List with:

- **overall_performance:** Data frame with Sensitivity, Specificity, FPR, FNR.
- **edge_significance:** Data frame with per-edge recovery rates and statistics.

individual_runs List with:

- **list_of_summaries:** Bootstrap summaries from each run.
- **list_of_per_edge_performance:** Per-edge results from each run.

successful_runs Number of successfully completed runs.

Examples

```
## Not run:
# First create a TNA model from some data
trans_mat <- matrix(c(
  0.6, 0.3, 0.1,
  0.2, 0.6, 0.2,
  0.1, 0.2, 0.7
), nrow = 3, byrow = TRUE)
rownames(trans_mat) <- colnames(trans_mat) <- c("A", "B", "C")

# Create a mock Model object
Model <- list(
  weights = trans_mat,
  inits = c(A = 0.33, B = 0.34, C = 0.33)
```

```

)

# Define stable transitions
stable <- list(c("A", "B"), c("B", "C"))

# Run bootstrap simulation
results <- run_bootstrap_simulation(
  Model = Model,
  stable_transitions = stable,
  num_runs = 50,
  seq_length = 30,
  n_sequences = 100,
  stability_prob = 0.95,
  unstable_mode = "random_jump",
  num_cores = 4
)

# View overall performance
results$aggregated_summary$overall_performance

# View edge-level results
results$aggregated_summary$edge_significance

# Old parameter names still work
results <- run_bootstrap_simulation(
  Model = Model,
  stable_transitions = stable,
  num_runs = 50,
  max_seq_length = 30,
  num_rows = 100
)

## End(Not run)

```

run_grid_simulation *Run Grid Search Over Simulation Parameters*

Description

Execute bootstrap simulations across a grid of parameter combinations. Useful for studying how different settings affect edge recovery performance.

Usage

```

run_grid_simulation(
  Model,
  stable_transitions,
  num_runs,
  n_sequences_vec = NULL,

```

```

seq_length_vec = NULL,
na_range_list = list(list(min = 0, max = 0)),
stability_prob = 0.95,
unstable_mode = "random_jump",
unstable_random_transition_prob = 0.5,
unstable_perturb_noise = 0.5,
unlikely_prob_threshold = 0.1,
include_na = TRUE,
consistency_range = c(0.75, 1.25),
level = 0.05,
num_cores = parallel::detectCores() - 1,
num_rows_vec = NULL,
max_seq_length_vec = NULL
)

```

Arguments

Model	A TNA model object with weights (transition matrix) and inits (initial probabilities).
stable_transitions	List of character vectors defining ground truth stable transitions.
num_runs	Integer. Number of bootstrap runs per parameter combination.
n_sequences_vec	Numeric vector. Values of n_sequences to test.
seq_length_vec	Numeric vector. Values of seq_length to test.
na_range_list	List of lists. Each inner list has min and max elements defining an NA range to test.
stability_prob	Numeric. Fixed stability probability. Default: 0.95.
unstable_mode	Character. Fixed unstable mode. Default: "random_jump".
unstable_random_transition_prob	Numeric. Fixed unstable probability. Default: 0.5.
unstable_perturb_noise	Numeric. Fixed perturbation noise. Default: 0.5.
unlikely_prob_threshold	Numeric. Fixed unlikely threshold. Default: 0.1.
include_na	Logical. Whether to include NAs. Default: TRUE.
consistency_range	Numeric vector of length 2. Bootstrap consistency range. Default: c(0.75, 1.25).
level	Numeric. Significance level. Default: 0.05.
num_cores	Integer. Number of cores for parallel processing. Default: detectCores() - 1.
num_rows_vec	Deprecated. Use n_sequences_vec instead.
max_seq_length_vec	Deprecated. Use seq_length_vec instead.

Details

The function creates a full factorial grid from:

- `n_sequences_vec` x `seq_length_vec` x `na_range_list`

For each combination, it runs `run_bootstrap_simulation()` and stores the results along with the parameter values used.

Progress messages are printed to track execution.

Value

A named list where each element corresponds to a parameter combination. Names follow the pattern `nr<n_sequences>_sl<seq_length>_na<min>-<max>`. Each element contains:

aggregated_summary Aggregated performance and edge significance.

individual_runs Per-run details.

successful_runs Number of successful runs.

parameters The parameter values used for this combination.

Examples

```
## Not run:
# Create a model
trans_mat <- matrix(c(
  0.6, 0.3, 0.1,
  0.2, 0.6, 0.2,
  0.1, 0.2, 0.7
), nrow = 3, byrow = TRUE)
rownames(trans_mat) <- colnames(trans_mat) <- c("A", "B", "C")

Model <- list(
  weights = trans_mat,
  inits = c(A = 0.33, B = 0.34, C = 0.33)
)

stable <- list(c("A", "B"), c("B", "C"))

# Run grid search
grid_results <- run_grid_simulation(
  Model = Model,
  stable_transitions = stable,
  num_runs = 20,
  n_sequences_vec = c(50, 100, 200),
  seq_length_vec = c(20, 30, 50),
  na_range_list = list(
    list(min = 0, max = 0),
    list(min = 0, max = 5),
    list(min = 5, max = 10)
  ),
  num_cores = 4
)
```

```
# Analyze results
summarize_grid_results(grid_results)

# Old parameter names still work
grid_results <- run_grid_simulation(
  Model = Model,
  stable_transitions = stable,
  num_runs = 20,
  num_rows_vec = c(50, 100, 200),
  max_seq_length_vec = c(20, 30, 50)
)

## End(Not run)
```

run_network_simulation

Run Network Analysis Simulations

Description

Run TNA simulations comparing fitted models to original data. Supports multiple model types, comparison modes, and parallel processing.

Usage

```
run_network_simulation(
  original_data_list,
  sim_params = NULL,
  models = c("tna"),
  comparisons = c("original"),
  num_runs = 3,
  parallel = FALSE,
  scaling = "minmax"
)
```

Arguments

original_data_list	A list of data frames containing original sequence data, or a single data frame. Used as reference for comparisons.
sim_params	Simulation parameters. Can be: • NULL (uses defaults). • A list of parameters. • A data frame where each row is a parameter combination.
models	Character vector of model types to simulate. Options: "tna", "ftna", "ctna", "atna". Default: c("tna").

comparisons	Character vector of comparison types. Options: "original" Compare simulated models to original data. "across_models" Compare different model types against each other. "ftna_reference" Compare all models to their ftna counterparts. Default: c("original").
num_runs	Integer. Number of simulation runs per parameter set. Default: 3.
parallel	Logical. Whether to use parallel processing. Default: FALSE.
scaling	Character. Scaling method for comparisons. Options: "none", "minmax", "zscore". Default: "minmax".

Details

For each parameter combination and original dataset, the function:

1. Extracts transition probabilities from the original TNA model.
2. Generates simulated sequences using those probabilities.
3. Fits the specified model types to the simulated sequences.
4. Computes comparison metrics against reference models.

Parallel processing uses the future.apply package when enabled.

Value

A list containing:

metrics Data frame of all comparison metrics from all runs.

summary_stats Aggregated summary statistics by model and comparison type.

parameters List of simulation parameters used.

Examples

```
## Not run:
# Create some original sequence data
trans_mat <- matrix(c(
  0.7, 0.2, 0.1,
  0.3, 0.5, 0.2,
  0.2, 0.3, 0.5
), nrow = 3, byrow = TRUE)
rownames(trans_mat) <- colnames(trans_mat) <- c("A", "B", "C")
init_probs <- c(A = 0.5, B = 0.3, C = 0.2)

original_data <- simulate_sequences(
  trans_matrix = trans_mat,
  init_probs = init_probs,
  seq_length = 30,
  n_sequences = 100
)
```

```
# Run simulations comparing tna and ftna models
results <- run_network_simulation(
  original_data_list = original_data,
  sim_params = list(seq_length = 30, n_sequences = 100),
  models = c("tna", "ftna"),
  comparisons = c("original"),
  num_runs = 5
)

# View summary statistics
results$summary_stats

## End(Not run)
```

run_sampling_analysis *Run Sampling Analysis on TNA Models*

Description

Perform comprehensive TNA model analysis through repeated sampling and statistical comparison. Uses a statistically correct sampling procedure by comparing models built on independent data subsets (sample vs remaining).

Usage

```
run_sampling_analysis(
  model,
  sampling_percent = 0.3,
  iterations = 100,
  seed = NULL,
  model_scaling = NULL,
  verbose = TRUE
)
```

Arguments

model	A TNA model object created with tna() or related functions.
sampling_percent	Numeric value between 0 and 1 specifying the proportion of data to use for the sample model. Default: 0.3.
iterations	Integer specifying the number of sampling iterations. Default: 100.
seed	Integer seed for reproducible random sampling. Default: NULL.
model_scaling	Character string to override the model's scaling: NULL (default) uses original model's scaling, "skip" forces no scaling, or valid TNA scaling options ("min-max", "max", "rank").
verbose	Logical. Print progress messages. Default: TRUE.

Details

This function implements a statistically correct sampling procedure:

1. **Data Splitting:** For each iteration, splits the original data into:
 - Sample set (specified percentage)
 - Remaining set (complement)
2. **Model Building:** Builds TNA models on both datasets using the original model's type and configurable scaling.
3. **Model Comparison:** Compares sample model vs remaining model using TNA's compare() function, providing metrics across categories:
 - Correlations (Pearson, Spearman, Kendall)
 - Dissimilarities (Euclidean, Manhattan, etc.)
 - Similarities (Cosine, Jaccard, etc.)
4. **Result Aggregation:** Collects metrics across all iterations and computes summary statistics (mean, sd, median, quartiles).

This approach is statistically superior to comparing sample vs original because it compares models built on independent data subsets, providing a true measure of sampling variability and model stability.

Value

A list containing:

aggregated Data frame with aggregated summary statistics by metric.

individual Data frame with raw metric values from each iteration.

params Parameters used for the analysis.

Examples

```
## Not run:
library(tna)
model <- tna(group_regulation)

# Basic analysis with defaults
results <- run_sampling_analysis(model, iterations = 50)
results$aggregated

# With custom parameters
results <- run_sampling_analysis(
  model,
  sampling_percent = 0.4,
  iterations = 100,
  seed = 42
)

## End(Not run)
```

run_scenario	<i>Run a Whole Scenario</i>
--------------	-----------------------------

Description

Execute every case of a scenario by `do.call()`-ing its argument-list against the scenario's simulator. Returns a named list of the resulting `saqr_sim` objects, labelled case1, case2, ...

Usage

```
run_scenario(scenario, seed = NULL)
```

Arguments

scenario	Character scalar. The scenario name (see list_scenarios).
seed	Integer scalar or NULL. When supplied, each case is given a deterministic seed of seed + case index so the whole run is reproducible while every case still differs. When NULL the simulators use their own defaults.

Value

A named list of `saqr_sim` objects, one per case, named case1, case2, ...

Examples

```
s <- run_scenario("power_ttest", seed = 1)
s$case1
```

sample_tna	<i>Sample and Re-estimate TNA Model</i>
------------	---

Description

Take a TNA model, sample a percentage of its data, and re-estimate a new TNA model from the sampled data using the same parameters.

Usage

```
sample_tna(model, sampling_percent = 0.3, model_type = NULL, scaling = NULL)
```

Arguments

model	A TNA model object or data frame containing sequence data.
sampling_percent	Numeric value between 0 and 1 indicating the proportion of data to sample. Default: 0.3.
model_type	Character string specifying the model type. If NULL, uses the type from the original model. Default: NULL.
scaling	Character string specifying the scaling method. If NULL, uses the scaling from the original model. Default: NULL.

Value

A new TNA model object estimated from the sampled data.

Examples

```
## Not run:
library(tna)
model <- tna(group_regulation)
sampled <- sample_tna(model, 0.3)

## End(Not run)
```

select_states	<i>Select States for Networks</i>
---------------	-----------------------------------

Description

Intelligently select learning states based on the number of nodes needed, optionally biasing toward specific categories.

Usage

```
select_states(
  n_states,
  primary_categories = NULL,
  secondary_categories = NULL,
  primary_ratio = 0.6,
  seed = NULL
)

smart_select_states(...)
```

Arguments

<code>n_states</code>	Integer. Number of states needed.
<code>primary_categories</code>	Character vector. Categories to prioritize. Default: NULL (balanced selection).
<code>secondary_categories</code>	Character vector. Categories to supplement with. Default: NULL (use all remaining).
<code>primary_ratio</code>	Numeric (0 to 1). Proportion of states from primary categories. Default: 0.6.
<code>seed</code>	Integer or NULL. Random seed. Default: NULL.
<code>...</code>	Arguments passed to <code>select_states</code> .

Details

Selection logic:

- If `n_states` $\leq$ 5: Single category or balanced small set
- If `n_states` 6-10: 1-2 categories prioritized
- If `n_states` 11-20: Multiple categories with primary focus
- If `n_states` $>$ 20: All categories combined

Value

Character vector of selected learning states.

Examples

```
# 5 states focused on self-regulation
select_states(5, primary_categories = "metacognitive")

# 10 states: mostly cognitive, some behavioral
select_states(10,
  primary_categories = "cognitive",
  secondary_categories = "behavioral"
)

# 15 states: balanced across categories
select_states(15, seed = 42)
```

simulate

*Simulate Data via a Unified Dispatcher***Description**

A single discoverable entry point over every explicit-parameter simulator in the package. You pass the simulation type plus the named arguments that simulator expects, and `simulate()` forwards them straight through, returning that simulator's `saqr_sim` object unchanged. Use [list_simulators](#) to see the available types.

For random, seed-driven generation (where the seed picks both the data and the structural parameters) and a bare `data.frame` return, use the separate [simulate_data](#) entry point instead.

Usage

```
simulate(type, ..., seed = NULL)
```

Arguments

<code>type</code>	Character scalar. The simulation type. One of the values in the <code>type</code> column of list_simulators : "ttest", "anova", "correlation", "clusters", "prediction", "regression", "lpa", "lca", "fa", "seq_clusters", "longitudinal", "mlm", "growth", "irt", "survival", "hmm".
<code>...</code>	Named arguments passed straight through to the matching simulator. See the help page of the underlying simulator (e.g. simulate_ttest) for the arguments it accepts.
<code>seed</code>	Integer or NULL. Random seed, forwarded to the target simulator.

Value

The `saqr_sim` object returned by the dispatched simulator (with `$data`, `$params`, `$type`, `$seed`).

See Also

[list_simulators](#) for the catalogue of types, [simulate_data](#) for random seed-driven generation.

Examples

```
simulate("ttest", n_a = 50, n_b = 50, mean_a = 0, mean_b = 0.5, seed = 1)

simulate("anova", n = 30, means = c(10, 12, 15), seed = 1)
```

simulate_anova

*Simulate Multi-Group Comparison Data (ANOVA)***Description**

Generate data for a one-way ANOVA with fully specified group means and standard deviations. Designed so that `aov(score ~ group, data = r$data)` recovers the true group effects at large n .

Usage

```
simulate_anova(n, means, sds = 1, labels = NULL, seed = NULL)
```

Arguments

<code>n</code>	Integer or integer vector. If a single value, the per-group sample size (equal groups). If a vector, must have length equal to <code>length(means)</code> giving per-group sizes.
<code>means</code>	Numeric vector. Population mean for each group.
<code>sds</code>	Numeric scalar or vector. Standard deviation(s) for each group. Scalar is recycled. Default: 1.
<code>labels</code>	Character vector or NULL. Group labels. Default: <code>paste0("G", seq_along(means))</code> .
<code>seed</code>	Integer or NULL. Random seed.

Value

A `saqr_sim` object with:

`$data` data.frame with columns `group` (factor) and `score` (numeric).

`$params` list with `means`, `sds` (as vector), `n` (per-group sizes), `labels`, `eta_squared` (true population η^2).

Examples

```
r <- simulate_anova(n = 30, means = c(10, 12, 15), seed = 1)
summary(aov(score ~ group, data = r$data))
r$params$eta_squared

# Unequal groups and heteroscedastic
r2 <- simulate_anova(n = c(50, 30, 20), means = c(5, 5, 10),
                     sds = c(1, 2, 3), seed = 42)
```

 simulate_correlation *Simulate Correlated Multivariate Data*

Description

Generate multivariate normal data from an explicit correlation (or covariance) matrix. Designed so that `cor(r$data)` recovers `sigma` at large `n`.

Usage

```
simulate_correlation(n, sigma, means = NULL, var_names = NULL, seed = NULL)
```

Arguments

<code>n</code>	Integer. Sample size.
<code>sigma</code>	Numeric matrix. Either a correlation matrix (all diagonal = 1) or a covariance matrix. Must be symmetric and positive-definite (or near-PD; corrected internally).
<code>means</code>	Numeric vector or NULL. Population means for each variable. Default: all zeros.
<code>var_names</code>	Character vector or NULL. Variable names. Default: <code>paste0("x", seq_len(ncol(sigma)))</code> .
<code>seed</code>	Integer or NULL. Random seed.

Value

A `saqr_sim` object with:

`$data` `data.frame` with columns `x1...xp`.

`$params` list with `sigma` (the input matrix), `means`, `is_correlation` (logical).

Examples

```
# Correlation matrix
R <- matrix(c(1, 0.6, 0.3,
              0.6, 1, 0.5,
              0.3, 0.5, 1), nrow = 3)
r <- simulate_correlation(n = 200, sigma = R, seed = 1)
cor(r$data)           # should approximate R
r$params$sigma        # the true matrix

# With means and custom names
r2 <- simulate_correlation(n = 500, sigma = R,
                           means = c(10, 20, 30),
                           var_names = c("IQ", "GPA", "income"), seed = 42)
```

simulate_data

*Simulate Ready-to-Use Statistical Datasets***Description**

Generate synthetic datasets suitable for common statistical analyses. Each dataset type contains real signal so the intended analysis works out of the box. Different seeds produce structurally different datasets (varying n, effect sizes, number of groups/variables).

Usage

```
simulate_data(type, seed = NULL, complexity = "clean", ..., n_batch = NULL)
```

Arguments

type	Character string specifying the dataset type. One of: "ttest" Two-group comparison. Columns: group (factor), score (numeric). "anova" Multi-group comparison (3–5 groups). Columns: group (factor), score (numeric). "correlation" Correlated variables (4–7 columns). All numeric x1–xp. "clusters" Cluster structure. Numeric x1–xd plus integer true_cluster. "factor_analysis" Latent factor structure. All numeric x1–xp. Attributes n_factors and loadings. "prediction" Regression dataset. Columns: y, x1–x4, cat1, cat2. "mlvar" Multilevel VAR panel data. Columns: id, day, beep, V1–Vd. Attributes true_temporal, true_contemporaneous, vars. "batch" Wildcard: generates all 7 types. Returns a named list with n_batch (default 1000) datasets per type.
seed	Integer or NULL. Random seed. Also determines structural parameters (n, effect sizes, etc.). Default: NULL.
complexity	Character. Controls edge-case injection for stress testing. "clean" (default for single datasets) No edge cases. "auto" (default in batch mode) Randomly injects 0–3 edge cases per dataset (seed-driven, fully reproducible). character vector Inject specific cases. One or more of: "na", "outliers", "ties", "duplicates", "constant_col", "all_na_col", "tiny_n", "heavy_tailed", "heteroscedastic", "extreme_imbalance", "multicollinear".
...	Optional overrides for structural parameters (n, n_groups, etc.).
n_batch	Integer or NULL. When provided, returns a list of n_batch datasets instead of a single dataset. When type = "batch", defaults to 1000L. Each item has seed, batch_id, and complexity attributes.

Value

A data.frame (single dataset) or a list (when n_batch is non-NULL or type = "batch"). Each dataset has attributes type, info, and complexity. Batch items additionally have seed and batch_id.

See Also

[simulate_sequences\(\)](#), [simulate_igraph\(\)](#)

Examples

```
# Single clean dataset
d <- simulate_data("ttest", seed = 42)

# With edge cases
d <- simulate_data("correlation", seed = 1, complexity = "auto")
d <- simulate_data("ttest", seed = 1, complexity = c("na", "outliers"))

# Batch of 100 ttest datasets
batch <- simulate_data("ttest", seed = 1, n_batch = 100)

# All types, 1000 datasets each
all_batches <- simulate_data("batch", seed = 1)
```

simulate_edge_list	<i>Simulate Social Network Edge List</i>
--------------------	--

Description

Generate a simulated edge list for social network analysis. Creates random connections between nodes with weights and class assignments.

Usage

```
simulate_edge_list(
  n_nodes = 20,
  n_edges = NULL,
  edge_density = 3,
  n_classes = 3,
  directed = TRUE,
  allow_self_loops = FALSE,
  weight_range = c(0.1, 1),
  names = NULL,
  class_probs = NULL,
  seed = NULL
)
```

Arguments

n_nodes	Integer. Number of nodes (people) in the network. Default: 20.
n_edges	Integer or NULL. Number of edges to generate. If NULL, calculated as $n_nodes * edge_density$. Default: NULL.
edge_density	Numeric. Average number of edges per node when n_edges is NULL. Default: 3.
n_classes	Integer. Number of classes/groups (2-10). Default: 3.
directed	Logical. Whether edges are directed. Default: TRUE.
allow_self_loops	Logical. Whether to allow self-connections. Default: FALSE.
weight_range	Numeric vector of length 2. Range for edge weights. Default: c(0.1, 1.0).
names	Character vector or NULL. Custom node names. If NULL, uses names from GLOBAL_NAMES. Default: NULL.
class_probs	Numeric vector or NULL. Probability of each class. Must sum to 1 and have length n_classes . If NULL, uniform distribution. Default: NULL.
seed	Integer or NULL. Random seed for reproducibility. Default: NULL.

Details

The function generates a random social network edge list with:

- Nodes named using diverse global names (or custom names)
- Random edges between nodes
- Weights uniformly distributed in the specified range
- Class assignments based on specified probabilities

For undirected networks, each edge appears once (no duplicate A-B, B-A pairs).

Value

A data frame with columns:

source Character. Name of the source node.
target Character. Name of the target node.
weight Numeric. Edge weight in the specified range.
class Integer. Class assignment (1 to **n_classes**).

See Also

[GLOBAL_NAMES](#), [get_global_names](#)

Examples

```
# Basic usage with defaults
edges <- simulate_edge_list(seed = 42)
head(edges)

# Larger network with 5 classes
edges <- simulate_edge_list(
  n_nodes = 50,
  n_classes = 5,
  edge_density = 4,
  seed = 123
)

# Custom names and specific number of edges
edges <- simulate_edge_list(
  n_nodes = 10,
  n_edges = 30,
  names = c("Alice", "Bob", "Carol", "Dave", "Eve",
            "Frank", "Grace", "Hank", "Ivy", "Jack"),
  n_classes = 2,
  seed = 42
)

# Undirected network with unequal class distribution
edges <- simulate_edge_list(
  n_nodes = 30,
  directed = FALSE,
  n_classes = 3,
  class_probs = c(0.5, 0.3, 0.2),
  seed = 42
)

# View class distribution
table(edges$class)
```

simulate_fa

Simulate Factor Analysis Data with Known Parameters

Description

Generate multivariate normal data from an explicit factor model for factor analysis parameter recovery testing. The implied covariance matrix $\Sigma = \Lambda\Phi\Lambda' + \Psi$ is fully specified and stored in the return value for direct comparison with estimated covariances.

Usage

```
simulate_fa(loadings, phi = NULL, psi = NULL, n, seed = NULL)
```

Arguments

loadings	Numeric matrix of dimension $p \times m$ (observed variables $\times$ factors). Must be a matrix (not a vector). Zero entries indicate no loading; use <code>matrix(val, nrow = p, ncol = 1)</code> for single-factor models.
phi	Numeric matrix of dimension $m \times m$. Factor correlation matrix. Must be symmetric and positive-definite. Defaults to <code>diag(m)</code> (orthogonal factors).
psi	Numeric vector of length p . Unique variances. All values must be positive. Defaults to <code>1 - diag(loadings %>% phi %>% t(loadings))</code> (communality complement). An error is raised if any auto-computed value is ≤ 0 (over-factored model).
n	Positive integer. Sample size.
seed	Integer or NULL. Random seed.

Value

A named list with elements:

`data` `data.frame` with columns `y1...yp`.

`params` list with `loadings`, `phi`, `psi`, and `sigma_implied` (the *pxp* model-implied covariance matrix).

Examples

```
# Two orthogonal factors, 6 indicators (3 per factor)
loadings <- matrix(c(0.8, 0.7, 0.6, 0, 0, 0,
                    0, 0, 0, 0.8, 0.7, 0.6),
                  nrow = 6, ncol = 2) # byrow = FALSE (default)
r <- simulate_fa(loadings = loadings, n = 500, seed = 1)
r$data          # data.frame: y1..y6
r$params$sigma_implied # implied covariance matrix

# Oblique model
phi <- matrix(c(1, 0.4, 0.4, 1), nrow = 2)
r2 <- simulate_fa(loadings = loadings, phi = phi, n = 500, seed = 1)
```

`simulate_group_tna_networks`

Simulate Group TNA Network Objects

Description

Simulate group TNA network objects (fitted `group_tna` models) for simulation studies with grouped/clustered data. Each network contains multiple groups (e.g., classrooms, teams) with their own transition patterns.

Usage

```
simulate_group_tna_networks(
  n_groups = 5,
  n_actors = 10,
  n_states = 8,
  seq_length_range = c(10, 30),
  use_learning_states = TRUE,
  categories = NULL,
  seed = NULL,
  verbose = TRUE,
  actors_per_group = NULL,
  min_seq_length = NULL,
  max_seq_length = NULL,
  learning_categories = NULL,
  ...
)

generate_group_tna_networks(...)
```

Arguments

<code>n_groups</code>	Integer. Number of groups in the network. Default: 5.
<code>n_actors</code>	Integer or integer vector. Number of actors per group. Accepts: single integer (fixed size), two integers like <code>c(8, 12)</code> (min/max), or a range like <code>5:15</code> (min/max taken from range). Default: 10.
<code>n_states</code>	Integer. Number of states/actions in the network. Default: 5.
<code>seq_length_range</code>	Integer vector of length 2. Range for sequence lengths per actor (min, max). Default: <code>c(10, 30)</code> .
<code>use_learning_states</code>	Logical. If TRUE, uses learning action verbs as state names. Default: TRUE.
<code>categories</code>	Character vector or NULL. Which categories of learning verbs to use. Options: "metacognitive", "cognitive", "behavioral", "social", "motivational", "affective", "group_regulation", or "all". If NULL (default), randomly selects one category.
<code>seed</code>	Integer or NULL. Random seed for reproducibility. Default: NULL.
<code>verbose</code>	Logical. If TRUE, prints progress messages. Default: TRUE.
<code>actors_per_group</code>	Deprecated. Use <code>n_actors</code> instead.
<code>min_seq_length</code>	Deprecated. Use <code>seq_length_range</code> instead.
<code>max_seq_length</code>	Deprecated. Use <code>seq_length_range</code> instead.
<code>learning_categories</code>	Deprecated. Use <code>categories</code> instead.
<code>...</code>	Arguments passed to <code>simulate_group_tna_networks</code> .

Details

This function generates a single group TNA network with multiple groups. It simulates long-format sequence data with group structure, converts it to wide format, and fits a group TNA model using `tna::group_model()`.

Use Cases:

- Simulating classroom-level learning behavior data
- Generating team collaboration sequences
- Creating multi-group datasets for method comparison

Value

A `group_tna` model object (class "group_tna") containing:

networks List of TNA networks, one per group

data The underlying wide-format data

group The grouping variable name

See Also

[simulate_tna_networks](#) for individual TNA models, [simulate_long_data](#) for generating long-format group data, [fit_network_model](#) for model fitting.

Examples

```
## Not run:
# Generate a group TNA network with 4 groups, 15 actors each
group_net <- simulate_group_tna_networks(
  n_groups = 4,
  n_actors = 15,
  n_states = 5,
  seed = 42
)

# Variable group sizes using range notation
var_net <- simulate_group_tna_networks(
  n_groups = 5,
  n_actors = c(8, 15),
  seq_length_range = c(5, 25),
  seed = 123
)

# With specific learning category
ssrl_net <- simulate_group_tna_networks(
  n_groups = 6,
  n_actors = c(10, 20),
  categories = "group_regulation",
  seed = 456
)
```

```
# Access individual group networks
names(group_net)
group_net[[1]]$weights

# Old parameter names still work
group_net <- simulate_group_tna_networks(
  actors_per_group = 10,
  min_seq_length = 5,
  max_seq_length = 20,
  seed = 42
)

## End(Not run)
```

simulate_growth

Simulate Latent Growth-Curve Data

Description

Generate longitudinal data from a linear latent growth-curve model. Each subject draws a random intercept and slope from a bivariate normal with specified means, standard deviations, and correlation, then

$$y_{it} = b_{0i} + b_{1i} t + e_{it},$$

with time coded $0 \dots (n_time - 1)$. The result is returned in long format. Designed so that the mean per-subject slope recovers slope_mean and the spread of per-subject intercepts recovers intercept_sd at large n.

Usage

```
simulate_growth(
  n = 200,
  n_time = 5,
  intercept_mean = 0,
  slope_mean = 1,
  intercept_sd = 1,
  slope_sd = 0.5,
  intercept_slope_cor = 0,
  residual_sd = 1,
  seed = NULL
)
```

Arguments

n Integer. Number of subjects. Default: 200.

n_time Integer ≥ 2 . Number of measurement occasions per subject. Default: 5.

intercept_mean Numeric. Mean of subject intercepts b_{0i} . Default: 0.

slope_mean	Numeric. Mean of subject slopes b_{1i} . Default: 1.
intercept_sd	Positive numeric. Standard deviation of subject intercepts. Default: 1.
slope_sd	Positive numeric. Standard deviation of subject slopes. Default: 0.5.
intercept_slope_cor	Numeric in $[-1, 1]$. Correlation between subject intercepts and slopes. Default: 0.
residual_sd	Positive numeric. Level-1 (occasion) residual standard deviation σ_e . Default: 1.
seed	Integer or NULL. Random seed.

Value

A `saqr_sim` object with:

`$data` long-format data.frame with columns `subject` (factor), `time` (numeric, $0 \dots n_time-1$), and `y` (numeric).

`$params` list with means (intercept/slope), sds (intercept/slope), correlation, `residual_sd`, `n_time`, `subject_intercepts` (true b_{0i}), and `subject_slopes` (true b_{1i}).

Examples

```
r <- simulate_growth(n = 300, n_time = 6, slope_mean = 2,
                    intercept_sd = 1.5, seed = 1)
print(r)
head(r)

# Correlated intercepts and slopes
r2 <- simulate_growth(n = 250, intercept_slope_cor = 0.5, seed = 42)
r2$params$correlation
```

simulate_hmm

Simulate Hidden Markov Model Sequences

Description

Generate observed symbol sequences from a discrete hidden Markov model (HMM). For each sequence a latent state path is drawn from the Markov chain defined by `init` (initial distribution) and `trans` (state transition matrix); each hidden state then emits an observed symbol according to the emission matrix. The true latent paths are returned in `$params$hidden_paths` for parameter-recovery testing.

Usage

```
simulate_hmm(
  n_sequences = 50,
  seq_length = 30,
  n_states = 2,
  n_symbols = 3,
  trans = NULL,
  emission = NULL,
  init = NULL,
  seed = NULL
)
```

Arguments

<code>n_sequences</code>	Integer. Number of sequences to simulate. Default: 50.
<code>seq_length</code>	Integer. Length (number of time points) of each sequence. Default: 30.
<code>n_states</code>	Integer. Number of hidden states. Default: 2.
<code>n_symbols</code>	Integer. Number of observable symbols. Default: 3.
<code>trans</code>	Numeric matrix (<code>n_states</code> x <code>n_states</code>) or NULL. Row- stochastic state transition matrix. When NULL, a diagonally dominant matrix (states persist) is auto-generated.
<code>emission</code>	Numeric matrix (<code>n_states</code> x <code>n_symbols</code>) or NULL. Row- stochastic emission matrix. When NULL, an auto-generated matrix where each state favours a distinct symbol is used.
<code>init</code>	Numeric vector (length <code>n_states</code>) or NULL. Initial state distribution. When NULL, a uniform distribution is used.
<code>seed</code>	Integer or NULL. Random seed.

Details

The within-sequence latent recursion is inherently sequential and is implemented with `Reduce(..., accumulate = TRUE)` over the time index; the work across sequences is vectorised with `Map/vapply`.

Value

A `saqr_sim` object with:

`$data` long-format data.frame with columns `sequence_id` (integer), `time` (integer, 1...`seq_length`), and `symbol` (integer observed symbol in 1...`n_symbols`).

`$params` list with `trans`, `emission`, `init`, `n_states`, `n_symbols`, and `hidden_paths` (an `n_sequences` x `seq_length` integer matrix of the TRUE latent state sequences).

Examples

```
r <- simulate_hmm(n_sequences = 100, seq_length = 40, seed = 1)
print(r)
r$params$trans
```

```

r$params$emission
head(as.data.frame(r))

# Explicit three-state model
tr <- matrix(c(0.8, 0.1, 0.1,
              0.1, 0.8, 0.1,
              0.1, 0.1, 0.8), nrow = 3, byrow = TRUE)
em <- matrix(c(0.7, 0.2, 0.1,
              0.1, 0.7, 0.2,
              0.2, 0.1, 0.7), nrow = 3, byrow = TRUE)
r2 <- simulate_hmm(n_sequences = 80, seq_length = 50, n_states = 3,
                 n_symbols = 3, trans = tr, emission = em, seed = 42)
summary(r2)

```

simulate_htna

Simulate HTNA/MLNA/MTNA Matrix with Node Types

Description

Generate a transition matrix with multiple node types for hierarchical (HTNA), multilevel (MLNA), or multi-type (MTNA) network analysis. By default creates a 25-node matrix (5 nodes x 5 types) using learning category names.

`simulate_htna()`, `simulate_mlna()`, and `simulate_mtna()` are aliases that produce identical output - use whichever name fits your analysis context.

Usage

```

simulate_htna(
  n_nodes = 5,
  n_types = 5,
  type_names = c("Metacognitive", "Cognitive", "Behavioral", "Social", "Motivational"),
  within_prob = 0.4,
  between_prob = 0.15,
  weight_range = c(0, 1),
  allow_self_loops = FALSE,
  categories = c("metacognitive", "cognitive", "behavioral", "social", "motivational"),
  seed = NULL
)

simulate_mlna(
  n_nodes = 5,
  n_types = 5,
  type_names = c("Metacognitive", "Cognitive", "Behavioral", "Social", "Motivational"),
  within_prob = 0.4,
  between_prob = 0.15,
  weight_range = c(0, 1),
  allow_self_loops = FALSE,

```

```

categories = c("metacognitive", "cognitive", "behavioral", "social", "motivational"),
seed = NULL
)

simulate_mtna(
  n_nodes = 5,
  n_types = 5,
  type_names = c("Metacognitive", "Cognitive", "Behavioral", "Social", "Motivational"),
  within_prob = 0.4,
  between_prob = 0.15,
  weight_range = c(0, 1),
  allow_self_loops = FALSE,
  categories = c("metacognitive", "cognitive", "behavioral", "social", "motivational"),
  seed = NULL
)

```

Arguments

<code>n_nodes</code>	Integer. Number of nodes per type. Default: 5.
<code>n_types</code>	Integer. Number of node types. Default: 5.
<code>type_names</code>	Character vector. Names for node types. Default uses learning categories: "Metacognitive", "Cognitive", "Behavioral", "Social", "Motivational".
<code>within_prob</code>	Numeric. Probability of edges within each type. Default: 0.4.
<code>between_prob</code>	Numeric. Probability of edges between types. Default: 0.15.
<code>weight_range</code>	Numeric vector of length 2. Range for edge weights. Default: c(0, 1).
<code>allow_self_loops</code>	Logical. Allow diagonal entries. Default: FALSE.
<code>categories</code>	Character vector. Learning state categories for node names. Can be single (same for all types) or one per type. Default: matches <code>type_names</code> .
<code>seed</code>	Integer or NULL. Random seed. Default: NULL.

Value

A list containing:

matrix Numeric transition matrix (rows sum to 1).

node_types Named list mapping type names to node names (for `plot_htna`/`plot_mlna`).

type_names Character vector of type names.

n_nodes_per_type Integer vector of node counts per type.

See Also

[simulate_matrix](#), [simulate_tna_matrix](#)

Examples

```
# Default: 5 types x 5 nodes = 25 node matrix
net <- simulate_htna(seed = 42)
net$matrix
net$node_types # List format for plot_htna/plot_mlna

# Custom number of types
net <- simulate_htna(n_nodes = 4, n_types = 3, seed = 42)

# Custom type names
net <- simulate_mlna(
  n_nodes = 6,
  type_names = c("Macro", "Meso", "Micro"),
  seed = 42
)

# Use with tna package:
# plot_htna(net$matrix, net$node_types, layout = "polygon")
# plot_mlna(net$matrix, layers = net$node_types)
```

simulate_igraph

Simulate igraph Network Object

Description

Generate an igraph network object using common graph algorithms with realistic node names from human names or learning states.

Usage

```
simulate_igraph(
  n = NULL,
  model = c("er", "ba", "ws", "sbm", "reg", "grg", "ff"),
  name_source = c("human", "states"),
  regions = "all",
  categories = "all",
  names = NULL,
  directed = FALSE,
  weighted = FALSE,
  weights = c(0.1, 1),
  p = 0.1,
  m = NULL,
  power = 1,
  m_ba = 2,
  nei = 2,
  p_rewire = 0.05,
  blocks = 3,
```

```

    p_within = 0.3,
    p_between = 0.05,
    k = 4,
    radius = 0.25,
    fw = 0.35,
    bw = 0.32,
    seed = NULL
)

```

Arguments

n	Integer or NULL. Number of nodes. If NULL (default), randomly selects between 20-50 nodes.
model	Character. Graph generation algorithm: • "er": Erdos-Renyi random graph • "ba": Barabasi-Albert scale-free network • "ws": Watts-Strogatz small-world network • "sbm": Stochastic Block Model (community structure) • "reg": Regular graph (fixed degree) • "grg": Geometric Random Graph (spatial) • "ff": Forest Fire (growing network) Default: "er".
name_source	Character. Source for node names: • "human": Culturally diverse human names from GLOBAL_NAMES • "states": Learning action verbs from LEARNING_STATES Default: "human".
regions	Character vector. Regions to sample human names from (only used when name_source = "human"). Can be specific regions (e.g., "arab", "east_asia"), shortcuts (e.g., "europe", "africa", "asia"), or "all". See list_name_regions . Default: "all".
categories	Character vector. Learning state categories (only used when name_source = "states"). Options: "metacognitive", "cognitive", "behavioral", "social", "motivational", "affective", "group_regulation", "lms", or "all". Default: "all".
names	Character vector or NULL. Custom node names. Overrides name_source if provided. Default: NULL.
directed	Logical. If TRUE, generate directed network. Default: FALSE.
weighted	Logical. If TRUE, add random edge weights. Default: FALSE.
weights	Numeric vector of length 2. Weight range [min, max]. Default: c(0.1, 1.0).
p	Numeric. Edge probability for Erdos-Renyi model. Default: 0.1.
m	Integer or NULL. Fixed number of edges for Erdos-Renyi. Overrides p if provided. Default: NULL.
power	Numeric. Attachment power for Barabasi-Albert. Default: 1.
m_ba	Integer. Edges per new vertex for Barabasi-Albert. Default: 2.

nei	Integer. Neighborhood size for Watts-Strogatz. Default: 2.
p_rewire	Numeric. Rewiring probability for Watts-Strogatz. Default: 0.05.
blocks	Integer. Number of blocks for SBM. Default: 3.
p_within	Numeric. Within-block edge probability for SBM. Default: 0.3.
p_between	Numeric. Between-block edge probability for SBM. Default: 0.05.
k	Integer. Degree for regular graphs. Default: 4.
radius	Numeric. Connection radius for geometric random graph. Default: 0.25.
fw	Numeric. Forward burning probability for Forest Fire. Default: 0.35.
bw	Numeric. Backward burning factor for Forest Fire. Default: 0.32.
seed	Integer or NULL. Random seed for reproducibility. Default: NULL.

Details

This function wraps igraph's graph generation algorithms and adds:

- Meaningful node names (human names or learning states)
- Optional edge weights
- Block/community attributes for SBM

The generated networks are suitable for social network analysis, teaching network concepts, or testing TNA methods.

Value

An igraph object with vertex names and optional edge weights.

See Also

[simulate_network](#) for statnet network objects, [simulate_matrix](#) for adjacency matrices, [simulate_tna_network](#) for fitted TNA models.

Examples

```
library(igraph)

# Default: Erdos-Renyi with human names from all regions
g <- simulate_igraph(n = 15, seed = 42)
V(g)$name # Diverse names like "Yuki", "Omar", "Priya"
plot(g)

# Names from specific regions
g_arab <- simulate_igraph(n = 15, regions = "arab", seed = 42)
V(g_arab)$name # Arab names

g_africa <- simulate_igraph(n = 20, regions = "africa", seed = 42)
V(g_africa)$name # African names from all sub-regions

# Use learning state names instead
```

```

g_states <- simulate_igraph(n = 10, name_source = "states", seed = 42)
V(g_states)$name # Action verbs like "Plan", "Monitor", "Evaluate"

# Scale-free network (Barabasi-Albert)
g_sf <- simulate_igraph(n = 50, model = "ba", m_ba = 2, seed = 42)
degree_distribution(g_sf)

# Small-world network (Watts-Strogatz)
g_sw <- simulate_igraph(n = 30, model = "ws", nei = 4, p_rewire = 0.1, seed = 42)

# Community structure (Stochastic Block Model)
g_sbm <- simulate_igraph(n = 30, model = "sbm", blocks = 3,
                        p_within = 0.5, p_between = 0.05, seed = 42)
V(g_sbm)$block # Community assignments

# Weighted network with custom names
g_custom <- simulate_igraph(
  n = 10,
  model = "er",
  weighted = TRUE,
  names = c("Alice", "Bob", "Carol", "Dave", "Eve",
            "Frank", "Grace", "Heidi", "Ivan", "Judy"),
  seed = 42
)
E(g_custom)$weight

```

simulate_irt

Simulate Item Response Theory (IRT) Data

Description

Generate item-response data under a known IRT model with ground-truth item and person parameters. Supports the 1PL (Rasch), 2PL, 3PL dichotomous models and the graded response model (GRM) for ordered polytomous items. Designed so that, at large `n_persons`, item proportion-correct recovers item difficulty (negatively) and person total score recovers true ability (positively).

Response probabilities are computed for the whole person-by-item grid in one vectorised pass (`outer()` / matrix ops) and responses are drawn with a single comparison against a uniform random matrix – no loops.

Usage

```

simulate_irt(
  n_persons = 500,
  n_items = 20,
  model = c("2PL", "1PL", "3PL", "GRM"),
  n_categories = 2,
  a = NULL,
  b = NULL,

```

```

    c = NULL,
    theta = NULL,
    seed = NULL
  )

```

Arguments

<code>n_persons</code>	Integer. Number of persons (rows). Default: 500.
<code>n_items</code>	Integer. Number of items (columns). Default: 20.
<code>model</code>	Character. One of "2PL", "1PL", "3PL", "GRM". Matched via match.arg .
<code>n_categories</code>	Integer. Number of response categories. Must be 2 for the dichotomous models (1PL/2PL/3PL) and > 2 for GRM. Default: 2.
<code>a</code>	Numeric vector of length <code>n_items</code> or NULL. Item discriminations. NULL auto-generates <code>rlnorm(meanlog = 0, sdlog = 0.3)</code> (values near 1). For 1PL, discrimination is fixed at 1 regardless of <code>a</code> .
<code>b</code>	Numeric vector of length <code>n_items</code> or NULL. Item difficulties. NULL auto-generates $N(0, 1)$. For GRM, <code>b</code> is the item location around which ordered category thresholds are centred.
<code>c</code>	Numeric vector of length <code>n_items</code> or NULL. Lower-asymptote (guessing) parameters. NULL auto-generates 0.2 for 3PL and 0 otherwise. Ignored (forced to 0) for 1PL/2PL/GRM.
<code>theta</code>	Numeric vector of length <code>n_persons</code> or NULL. True person abilities. NULL auto-generates $N(0, 1)$.
<code>seed</code>	Integer or NULL. Random seed. <code>set.seed()</code> is called only when not NULL.

Value

A `saqr_sim` object with:

`$data` data.frame with `n_persons` rows and columns `item1...itemJ`. Values are 0/1 for dichotomous models and 0...`n_categories - 1` for GRM.

`$params` list with `model`, `a`, `b`, `c` (as used), `theta` (true abilities), `thresholds` (GRM category thresholds matrix, NULL otherwise), and `n_categories`.

Examples

```

# 2PL with default auto-generated parameters
fit <- simulate_irt(n_persons = 500, n_items = 20, model = "2PL", seed = 1)
print(fit)
fit$params$b

# Recover difficulty: easier items are answered correctly more often
prop_correct <- colMeans(as.data.frame(fit))
cor(prop_correct, fit$params$b)      # strongly negative

# Recover ability: higher total score tracks higher true theta
total <- rowSums(as.data.frame(fit))
cor(total, fit$params$theta)        # strongly positive

```

```
# Rasch (1PL): discrimination fixed at 1
simulate_irt(n_persons = 300, n_items = 10, model = "1PL", seed = 2)

# Graded response model with 4 ordered categories
simulate_irt(n_persons = 400, n_items = 8, model = "GRM",
             n_categories = 4, seed = 3)
```

simulate_lca

Simulate Latent Class Analysis Data

Description

Generate synthetic binary-indicator data with known latent class structure. All item response probabilities are explicitly specified.

Usage

```
simulate_lca(item_probs, class_probs, n, seed = NULL)
```

Arguments

item_probs	Numeric matrix of dimension $n_items \times n_classes$. Each column gives the probability that each item equals 1 for that class. All values must be in $[0, 1]$.
class_probs	Numeric vector of length $n_classes$. Class mixing proportions. Need not sum to 1 – normalised internally.
n	Positive integer. Total sample size.
seed	Integer or NULL. Random seed for reproducibility.

Value

A named list with elements:

data data.frame with binary columns `item1...itemm` and integer column `true_class` (labels 1...K).

params list with `item_probs` and `class_probs` (as supplied, un-normalised).

Examples

```
item_probs <- matrix(c(0.9, 0.1, 0.9, 0.1,
                      0.1, 0.9, 0.1, 0.9), nrow = 4, ncol = 2)
r <- simulate_lca(item_probs = item_probs, class_probs = c(0.5, 0.5),
                  n = 200, seed = 1)
r$data          # data.frame
r$params        # ground-truth parameters
```

simulate_long_data	<i>Simulate Long Format Sequence Data</i>
--------------------	---

Description

Generate simulated sequence data in long format, matching the structure of `group_regulation_long` in the `tna` package. Creates realistic educational data with actors nested in groups, groups nested in courses.

Usage

```
simulate_long_data(
  n_groups = 5,
  n_actors = 10,
  n_courses = 3,
  n_states = 9,
  states = NULL,
  use_learning_states = TRUE,
  categories = "group_regulation",
  seq_length_range = c(10, 30),
  achiever_levels = c("High", "Low"),
  achiever_probs = c(0.5, 0.5),
  start_time = "2025-01-01 10:00:00",
  time_interval_range = c(60, 600),
  trans_matrix = NULL,
  init_probs = NULL,
  seed = NULL,
  actors_per_group = NULL,
  actions = NULL,
  action_categories = NULL,
  n_actions = NULL,
  transition_probs = NULL,
  initial_probs = NULL
)
```

Arguments

<code>n_groups</code>	Integer. Number of groups. Default: 5.
<code>n_actors</code>	Integer or integer vector of length 2. If single integer, exact number of actors per group. If vector <code>c(min, max)</code> , random sizes in that range. Default: 10.
<code>n_courses</code>	Integer or character vector. Number of courses or specific course names (e.g., <code>c("A", "B", "C")</code>). Groups are distributed evenly across courses. Default: 3.
<code>n_states</code>	Integer. Number of actions to sample when using categories. Ignored if states is provided. Default: 9.
<code>states</code>	Character vector. The action/state names to use. If <code>NULL</code> , uses learning states based on categories. Default: <code>NULL</code> .

use_learning_states	Logical. If TRUE and states is NULL, uses learning state verbs from the specified categories. Default: TRUE.
categories	Character vector. Categories of learning states to use when states = NULL. Options: "metacognitive", "cognitive", "behavioral", "social", "motivational", "affective", "group_regulation", or "all". Default: "group_regulation".
seq_length_range	Integer vector of length 2. Range for sequence lengths per actor (min, max). Default: c(10, 30).
achiever_levels	Character vector. Levels for the Achiever variable. Default: c("High", "Low").
achiever_probs	Numeric vector. Probabilities for each achiever level. Must sum to 1. Default: c(0.5, 0.5).
start_time	POSIXct or character. Start time for timestamps. Default: "2025-01-01 10:00:00".
time_interval_range	Numeric vector of length 2. Range for time intervals between actions in seconds (min, max). Default: c(60, 600).
trans_matrix	Matrix or NULL. Custom transition probability matrix. If NULL, generates random probabilities. Default: NULL.
init_probs	Numeric vector or NULL. Custom initial probabilities. If NULL, generates random probabilities. Default: NULL.
seed	Integer or NULL. Random seed for reproducibility. Default: NULL.
actors_per_group	Deprecated. Use n_actors instead.
actions	Deprecated. Use states instead.
action_categories	Deprecated. Use categories instead.
n_actions	Deprecated. Use n_states instead.
transition_probs	Deprecated. Use trans_matrix instead.
initial_probs	Deprecated. Use init_probs instead.

Details

The data structure follows a nested hierarchy:

- **Courses** contain multiple **Groups**
- **Groups** contain multiple **Actors** (e.g., 10 per group)
- **Actors** have multiple **Actions** over time

This matches the structure of group_regulation_long from the tna package:

- Each group has a fixed number of actors
- Groups are distributed across courses
- Actors within a group share the same course

Built-in Group Regulation Actions: When `categories = "group_regulation"`, uses the 9 SSRL (Socially Shared Regulation of Learning) actions: adapt, cohesion, consensus, coregulate, discuss, emotion, monitor, plan, synthesis.

Value

A tibble (data frame) with columns:

Actor Integer. Actor identifier (1 to `n_actors`).

Achiever Character. Achievement level (e.g., "High", "Low").

Group Numeric. Group identifier (1 to `n_groups`).

Course Character. Course identifier.

Time POSIXct. Timestamp of the action.

Action Character. The action/state performed.

See Also

[simulate_onehot_data](#) for one-hot encoded format, [simulate_sequences](#) for generating wide-format sequences, [get_learning_states](#) for available learning states, [simulate_tna_networks](#) for generating TNA network objects.

Examples

```
## Not run:
# Basic usage: 5 groups, 10 actors each, 3 courses (new defaults)
data <- simulate_long_data(seed = 42)

# Explicit new parameter names
data <- simulate_long_data(
  n_groups = 10,
  n_actors = 15,
  n_states = 6,
  categories = c("metacognitive", "cognitive"),
  seq_length_range = c(15, 40),
  seed = 42
)

# Check structure
length(unique(data$Actor)) # 50 actors
length(unique(data$Group)) # 5 groups
table(table(data$Actor, data$Group) > 0) # 10 actors per group

# Variable group sizes
data <- simulate_long_data(
  n_groups = 30,
  n_actors = c(8, 12),
  states = c("Read", "Write", "Discuss", "Plan", "Review"),
  seed = 456
)
```

```
# Using specific learning state categories
data <- simulate_long_data(
  n_groups = 50,
  n_actors = 10,
  categories = c("metacognitive", "cognitive"),
  n_states = 8,
  seed = 789
)

# Old parameter names still work (backward compatible)
data <- simulate_long_data(
  actors_per_group = 10,
  actions = c("A", "B", "C"),
  seed = 42
)

## End(Not run)
```

simulate_longitudinal *Simulate Longitudinal Panel Data*

Description

Generate synthetic multilevel time-series data suitable for testing longitudinal models: multilevel VAR (mlVAR), random-intercept cross-lagged panel models (RI-CLPM), latent growth curves, and similar.

The data-generating process is a VAR(1) model with three layers:

Temporal Autoregressive and cross-lagged effects via matrix B. At each time point, $y(t) = \mu_i + B(y(t-1) - \mu_i) + \text{innovation}$.

Contemporaneous Within-time correlations among innovations, given by the Cholesky of contemporaneous.

Between-person Person-specific means μ_i drawn from $N(\text{grand_means}, \text{between})$.

Usage

```
simulate_longitudinal(
  n = 50L,
  tp = 50L,
  vars = 4L,
  temporal = NULL,
  contemporaneous = NULL,
  between = NULL,
  grand_means = NULL,
  innovation_sd = 1,
  beeps_per_day = NULL,
  ar_range = c(0.2, 0.5),
  cross_range = c(0.05, 0.2),
```

```

    n_cross = NULL,
    complexity = "clean",
    seed = NULL
)

```

Arguments

n	Integer. Number of subjects (persons). Default 50.
tp	Integer. Number of time points per subject. Default 50.
vars	Integer or character vector. If integer, the number of variables (named V1...Vp). If character, used as variable names directly. Default 4.
temporal	Numeric matrix (p x p). The VAR(1) coefficient matrix B. Element [i, j] is the effect of variable j at t-1 on variable i at t. Eigenvalues should have modulus < 1 for stationarity. Default: diagonal 0.3 (autoregressive only, no cross-lags).
contemporaneous	Numeric matrix (p x p). Correlation matrix of the innovation terms (within-time, within-person). Must be symmetric and positive-definite. Default: identity (uncorrelated innovations).
between	Numeric matrix (p x p). Covariance matrix of the person-specific means. Controls between-person variability. Default: identity.
grand_means	Numeric vector of length p. Population-level means for each variable. Default: all zeros.
innovation_sd	Numeric scalar or vector of length p. Standard deviation(s) of the innovation terms (before applying contemporaneous correlation). Default: 1.
beeps_per_day	Integer or NULL. If non-NULL, adds day and beep columns (ESM/EMA structure). tp must be divisible by beeps_per_day. Default: NULL (single time column).
ar_range	Numeric vector of length 2. When temporal is NULL, diagonal (autoregressive) coefficients are drawn uniformly from this range. Default: c(0.2, 0.5).
cross_range	Numeric vector of length 2. When temporal is NULL, off-diagonal (cross-lag) coefficients are drawn uniformly from this range (with random sign). Default: c(0.05, 0.2).
n_cross	Integer. When temporal is NULL, the number of non-zero cross-lagged effects to include. Default: min(3, p*(p-1)).
complexity	Character. Edge-case injection, same semantics as simulate_data . One of "clean" (default), "auto", or a character vector of specific cases (e.g. c("na", "outliers")). Supported cases: "na", "outliers", "heavy_tailed", "heteroscedastic", "tiny_n".
seed	Integer or NULL. Random seed.

Details

The temporal matrix B must be stationary (all eigenvalues inside the unit circle). If an auto-generated or user-supplied B violates this, it is rescaled by $0.95 / \max(\text{Mod}(\text{eigen}(B)\$values))$.

For ESM-style data with beeps_per_day, the first observation of each day is treated as a new starting point (no carry-over from previous day's last beep), matching the assumption in most ESM-VAR software.

Value

A `saqr_sim` object with:

`$data` data.frame with columns `id`, `time` (or `day/beep`), and `V1...Vp`.

`$params` list with `temporal` (B matrix), `contemporaneous` (innovation correlation), `between` (person-mean covariance), `grand_means`, `innovation_sd`, `n`, `tp`, `var_names`.

Examples

```
# Basic: 30 subjects, 60 time points, 3 variables
r <- simulate_longitudinal(n = 30, tp = 60, vars = 3, seed = 42)
head(r$data)
r$params$temporal # the true VAR(1) matrix

# Explicit temporal structure
B <- matrix(c(0.4, 0.0, 0.1,
              0.2, 0.3, 0.0,
              0.0, -0.1, 0.5), nrow = 3, byrow = TRUE)
r2 <- simulate_longitudinal(n = 50, tp = 100, vars = 3, temporal = B, seed = 1)

# ESM structure with days and beeps
r3 <- simulate_longitudinal(n = 40, tp = 70, vars = 4,
                           beeps_per_day = 7, seed = 7)
head(r3$data) # has day + beep columns

# With edge-case injection
r4 <- simulate_longitudinal(n = 30, tp = 50, vars = 3,
                           complexity = c("na", "outliers"), seed = 5)
```

simulate_lpa	<i>Simulate Latent Profile Analysis Data</i>
--------------	--

Description

Generate synthetic continuous-indicator data with known latent profile structure. All distributional parameters are explicitly specified so that parameter recovery can be verified in tests.

Usage

```
simulate_lpa(means, sds, props, n, seed = NULL)
```

Arguments

- `means` Numeric matrix of dimension `n_vars` x `n_profiles`. Each column is the vector of variable means for that profile.
- `sds` Numeric matrix of dimension `n_vars` x `n_profiles`, or a scalar / vector recycled to that shape. Standard deviations within each profile for each variable. All values must be positive.

props	Numeric vector of length n_profiles. Mixing proportions. Need not sum to 1 – normalised internally.
n	Positive integer. Total sample size.
seed	Integer or NULL. Random seed for reproducibility.

Value

A named list with elements:

data data.frame with columns y1...yp and true_profile (integer 1...K).

params list with means, sds (full matrix), and props (as supplied, un-normalised).

Examples

```
means <- matrix(c(0, 0, 10, 10), nrow = 2, ncol = 2)
r <- simulate_lpa(means = means, sds = 0.5, props = c(0.5, 0.5), n = 200, seed = 1)
r$data      # data.frame
r$params    # ground-truth parameters
```

simulate_matrix	<i>Simulate Network Matrix</i>
-----------------	--------------------------------

Description

Generate a simple transition matrix for TNA/Markov network analysis. Each call randomly selects a learning category for node names.

Usage

```
simulate_matrix(
  n_nodes = 9,
  matrix_type = c("transition", "frequency", "co-occurrence", "adjacency"),
  edge_prob = 0.3,
  weighted = TRUE,
  weight_range = c(0, 1),
  directed = TRUE,
  allow_self_loops = FALSE,
  names = NULL,
  seed = NULL
)
```

Arguments

n_nodes	Integer. Number of nodes. Default: 9.
matrix_type	Character. Type of matrix to generate: • "transition": Directed, row-normalized (rows sum to 1) • "frequency": Directed, integer counts • "co-occurrence": Symmetric • "adjacency": Binary or weighted edges Default: "transition".
edge_prob	Numeric. Probability of edges existing. Default: 0.3.
weighted	Logical. If TRUE, generates weighted edges. Default: TRUE.
weight_range	Numeric vector of length 2. Range for edge weights. Default: c(0, 1).
directed	Logical. If TRUE, generates directed matrix. Default: TRUE.
allow_self_loops	Logical. If TRUE, allows diagonal entries. Default: FALSE.
names	Character vector or NULL. Custom node names. If NULL, randomly selects learning states from a random category. Default: NULL.
seed	Integer or NULL. Random seed for reproducibility. Default: NULL.

Details

This function generates a simple network matrix. Each call randomly picks one learning category (metacognitive, cognitive, behavioral, social, motivational, affective, or group_regulation) and uses verbs from that category as node names.

For matrices with multiple node types, use [simulate_htna](#).

Value

A numeric matrix with row and column names set to learning states.

See Also

[simulate_htna](#) for multi-type matrices

Examples

```
# Simple 9-node transition matrix
mat <- simulate_matrix(seed = 42)
mat
rowSums(mat) # Rows sum to 1

# Frequency matrix
mat <- simulate_matrix(n_nodes = 5, matrix_type = "frequency", seed = 42)

# Co-occurrence matrix (symmetric)
mat <- simulate_matrix(n_nodes = 6, matrix_type = "co-occurrence", seed = 42)
isSymmetric(mat) # TRUE
```

```
# Custom names
mat <- simulate_matrix(n_nodes = 4, names = c("A", "B", "C", "D"), seed = 42)
```

simulate_mlm

Simulate Two-Level Multilevel (Hierarchical) Data

Description

Generate data for a two-level model where level-1 units are nested within clusters. The outcome follows

$$y_{ij} = (\beta_0 + u_{0j}) + \sum_k \beta_k x_{kij} + (u_{1j} x_{1ij}) + e_{ij},$$

where the first predictor carries the fixed slope, cluster random intercepts $u_{0j} \sim N(0, \tau_{00})$ with $\tau_{00} = \frac{icc}{1-icc} \sigma_e^2$, and (optionally) a cluster random slope $u_{1j} \sim N(0, \text{slope_sd}^2)$ on the first predictor. Designed so that the intraclass correlation and fixed slope are recovered at large `n_clusters`.

Usage

```
simulate_mlm(
  n_clusters = 30,
  cluster_size = 20,
  intercept = 0,
  slope = 0.5,
  residual_sd = 1,
  icc = 0.1,
  n_predictors = 1,
  random_slope = FALSE,
  slope_sd = 0,
  seed = NULL
)
```

Arguments

<code>n_clusters</code>	Integer. Number of level-2 clusters. Default: 30.
<code>cluster_size</code>	Integer scalar or integer vector of length <code>n_clusters</code> . Number of level-1 units per cluster. A vector gives an unbalanced design. Default: 20.
<code>intercept</code>	Numeric. Fixed intercept β_0 . Default: 0.
<code>slope</code>	Numeric. Fixed slope β_1 on the first predictor (x1). Default: 0.5.
<code>residual_sd</code>	Positive numeric. Level-1 residual standard deviation σ_e . Default: 1.
<code>icc</code>	Numeric in $[0, 1)$. Target intraclass correlation; used to derive the random-intercept variance $\tau_{00} = \frac{icc}{1-icc} \sigma_e^2$. Default: 0.1.
<code>n_predictors</code>	Integer ≥ 1 . Number of level-1 predictors (x1..xK). The first carries slope; the remaining predictors get fixed coefficients drawn from $N(0, 1)$. Default: 1.

random_slope	Logical. If TRUE, add a cluster-level random slope on the first predictor. Default: FALSE.
slope_sd	Non-negative numeric. Standard deviation of the random slope u_{1j} (used only when random_slope = TRUE). Default: 0.
seed	Integer or NULL. Random seed.

Value

A `saqr_sim` object with:

`$data` data.frame with columns `cluster_id` (factor), `y` (numeric), and `x1..xK` (numeric predictors).

`$params` list with `intercept`, `slope`, `betas` (full fixed-effect vector), `icc`, `tau00`, `residual_sd`, `slope_sd`, `random_slope`, `cluster_size`, `cluster_intercepts` (true u_{0j}), and `cluster_slopes` (true u_{1j} , NULL unless random_slope).

Examples

```
r <- simulate_mlm(n_clusters = 40, cluster_size = 25, slope = 0.8,
                 icc = 0.2, seed = 1)
print(r)
r$params$tau00

# Unbalanced clusters with a random slope
r2 <- simulate_mlm(n_clusters = 30, cluster_size = sample(10:30, 30, TRUE),
                  random_slope = TRUE, slope_sd = 0.3, seed = 42)
summary(r2)
```

simulate_network	<i>Simulate statnet Network Object</i>
------------------	--

Description

Generate a statnet network object using common graph algorithms with realistic node names from human names or learning states.

Usage

```
simulate_network(
  n = NULL,
  model = c("er", "ba", "ws", "sbm", "reg", "grg", "ff"),
  name_source = c("human", "states"),
  regions = "all",
  categories = "all",
  names = NULL,
  directed = FALSE,
  weighted = FALSE,
  weights = c(0.1, 1),
```

```

    p = 0.1,
    m = NULL,
    power = 1,
    m_ba = 2,
    nei = 2,
    p_rewire = 0.05,
    blocks = 3,
    p_within = 0.3,
    p_between = 0.05,
    k = 4,
    radius = 0.25,
    fw = 0.35,
    bw = 0.32,
    seed = NULL
  )

```

Arguments

n	Integer or NULL. Number of nodes. If NULL (default), randomly selects between 20-50 nodes.
model	Character. Graph generation algorithm: • "er": Erdos-Renyi random graph • "ba": Barabasi-Albert scale-free network • "ws": Watts-Strogatz small-world network • "sbm": Stochastic Block Model (community structure) • "reg": Regular graph (fixed degree) • "grg": Geometric Random Graph (spatial) • "ff": Forest Fire (growing network) Default: "er".
name_source	Character. Source for node names: • "human": Culturally diverse human names from GLOBAL_NAMES • "states": Learning action verbs from LEARNING_STATES Default: "human".
regions	Character vector. Regions to sample human names from (only used when name_source = "human"). Can be specific regions (e.g., "arab", "east_asia"), shortcuts (e.g., "europe", "africa", "asia"), or "all". See list_name_regions . Default: "all".
categories	Character vector. Learning state categories (only used when name_source = "states"). Options: "metacognitive", "cognitive", "behavioral", "social", "motivational", "affective", "group_regulation", "lms", or "all". Default: "all".
names	Character vector or NULL. Custom node names. Overrides name_source if provided. Default: NULL.
directed	Logical. If TRUE, generate directed network. Default: FALSE.
weighted	Logical. If TRUE, add random edge weights. Default: FALSE.
weights	Numeric vector of length 2. Weight range [min, max]. Default: c(0.1, 1.0).

p	Numeric. Edge probability for Erdos-Renyi model. Default: 0.1.
m	Integer or NULL. Fixed number of edges for Erdos-Renyi. Overrides p if provided. Default: NULL.
power	Numeric. Attachment power for Barabasi-Albert. Default: 1.
m_ba	Integer. Edges per new vertex for Barabasi-Albert. Default: 2.
nei	Integer. Neighborhood size for Watts-Strogatz. Default: 2.
p_rewire	Numeric. Rewiring probability for Watts-Strogatz. Default: 0.05.
blocks	Integer. Number of blocks for SBM. Default: 3.
p_within	Numeric. Within-block edge probability for SBM. Default: 0.3.
p_between	Numeric. Between-block edge probability for SBM. Default: 0.05.
k	Integer. Degree for regular graphs. Default: 4.
radius	Numeric. Connection radius for geometric random graph. Default: 0.25.
fw	Numeric. Forward burning probability for Forest Fire. Default: 0.35.
bw	Numeric. Backward burning factor for Forest Fire. Default: 0.32.
seed	Integer or NULL. Random seed for reproducibility. Default: NULL.

Details

This function generates networks using igraph algorithms internally, then converts to statnet's network class. The resulting object is compatible with all sna and network package functions.

Vertex names are stored in the "vertex.names" attribute and can be accessed with `network$vertex.names()`.

Value

A network object (class "network") with vertex names and optional edge weights.

See Also

[simulate_igraph](#) for igraph objects, [simulate_matrix](#) for adjacency matrices, [simulate_edge_list](#) for edge list data frames.

Examples

```
## Not run:
library(network)
library(sna)

# Default: Erdos-Renyi with human names
net <- simulate_network(n = 20, seed = 42)
class(net) # "network"
network$vertex.names(net) # Diverse human names

# Names from specific regions
net_asia <- simulate_network(n = 20, regions = "asia", seed = 42)
network$vertex.names(net_asia) # Asian names
```

```

# Use learning state names instead
net_states <- simulate_network(n = 15, name_source = "states", seed = 42)
network.vertex.names(net_states) # Action verbs

# Scale-free for SNA analysis
net_sf <- simulate_network(n = 50, model = "ba", seed = 42)
betweenness(net_sf)
closeness(net_sf)

# Community structure
net_sbm <- simulate_network(n = 30, model = "sbm", blocks = 3, seed = 42)

# Weighted network
net_w <- simulate_network(n = 20, model = "er", weighted = TRUE, seed = 42)
net_w %e% "weight" # Edge weights

## End(Not run)

```

simulate_onehot_data	<i>Simulate One-Hot Encoded Sequence Data</i>
----------------------	---

Description

Generate simulated sequence data in one-hot encoded format. Creates the same hierarchical structure as [simulate_long_data](#) (Actor, Achiever, Group, Course, Time) but with binary 0/1 columns for each state instead of a categorical Action column.

Usage

```

simulate_onehot_data(
  n_groups = 5,
  n_actors = 10,
  n_courses = 3,
  n_states = 9,
  states = NULL,
  use_learning_states = TRUE,
  categories = "group_regulation",
  seq_length_range = c(10, 30),
  achiever_levels = c("High", "Low"),
  achiever_probs = c(0.5, 0.5),
  start_time = "2025-01-01 10:00:00",
  time_interval_range = c(60, 600),
  trans_matrix = NULL,
  init_probs = NULL,
  sort_states = FALSE,
  state_prefix = "",
  seed = NULL,

```

```

    actors_per_group = NULL,
    actions = NULL,
    action_categories = NULL,
    n_actions = NULL,
    transition_probs = NULL,
    initial_probs = NULL
  )

```

Arguments

<code>n_groups</code>	Integer. Number of groups. Default: 5.
<code>n_actors</code>	Integer or integer vector of length 2. If single integer, exact number of actors per group. If vector <code>c(min, max)</code> , random sizes in that range. Default: 10.
<code>n_courses</code>	Integer or character vector. Number of courses or specific course names (e.g., <code>c("A", "B", "C")</code>). Groups are distributed evenly across courses. Default: 3.
<code>n_states</code>	Integer. Number of actions to sample when using categories. Ignored if states is provided. Default: 9.
<code>states</code>	Character vector. The action/state names to use. If NULL, uses learning states based on categories. Default: NULL.
<code>use_learning_states</code>	Logical. If TRUE and states is NULL, uses learning state verbs from the specified categories. Default: TRUE.
<code>categories</code>	Character vector. Categories of learning states to use when states = NULL. Options: "metacognitive", "cognitive", "behavioral", "social", "motivational", "affective", "group_regulation", or "all". Default: "group_regulation".
<code>seq_length_range</code>	Integer vector of length 2. Range for sequence lengths per actor (min, max). Default: <code>c(10, 30)</code> .
<code>achiever_levels</code>	Character vector. Levels for the Achiever variable. Default: <code>c("High", "Low")</code> .
<code>achiever_probs</code>	Numeric vector. Probabilities for each achiever level. Must sum to 1. Default: <code>c(0.5, 0.5)</code> .
<code>start_time</code>	POSIXct or character. Start time for timestamps. Default: "2025-01-01 10:00:00".
<code>time_interval_range</code>	Numeric vector of length 2. Range for time intervals between actions in seconds (min, max). Default: <code>c(60, 600)</code> .
<code>trans_matrix</code>	Matrix or NULL. Custom transition probability matrix. If NULL, generates random probabilities. Default: NULL.
<code>init_probs</code>	Numeric vector or NULL. Custom initial probabilities. If NULL, generates random probabilities. Default: NULL.
<code>sort_states</code>	Logical. Sort state columns alphabetically. Default: FALSE.
<code>state_prefix</code>	Character. Prefix for state column names. Default: "".
<code>seed</code>	Integer or NULL. Random seed for reproducibility. Default: NULL.
<code>actors_per_group</code>	Deprecated. Use <code>n_actors</code> instead.

actions	Deprecated. Use states instead.
action_categories	Deprecated. Use categories instead.
n_actions	Deprecated. Use n_states instead.
transition_probs	Deprecated. Use trans_matrix instead.
initial_probs	Deprecated. Use init_probs instead.

Details

This function internally calls [simulate_long_data](#) to generate the base data, then applies an internal one-hot helper to convert the Action column to one-hot encoded columns.

Each row will have exactly one state column with value 1, and all others with value 0, representing the action performed at that time point.

Value

A tibble (data frame) with columns:

Actor Integer. Actor identifier (1 to n_actors).

Achiever Character. Achievement level (e.g., "High", "Low").

Group Numeric. Group identifier (1 to n_groups).

Course Character. Course identifier.

Time POSIXct. Timestamp of the action.

Integer (0/1). One column per state, with 1 indicating the active state for that row.

See Also

[simulate_long_data](#) for categorical Action format, `Nestimate::action_to_onehot()` for converting existing long data, [simulate_sequences](#) for generating wide-format sequences.

Examples

```
## Not run:
# Basic usage with new standardized params
data <- simulate_onehot_data(n_groups = 5, n_actors = 3, seed = 42)
head(data)

# Custom states with prefix
data <- simulate_onehot_data(
  n_groups = 3,
  states = c("Read", "Write", "Think"),
  state_prefix = "state_",
  seed = 123
)
names(data)

# Verify one-hot encoding (each row sums to 1)
```

```

state_cols <- setdiff(names(data), c("Actor", "Achiever", "Group", "Course", "Time"))
all(rowSums(data[, state_cols]) == 1) # TRUE

# Old parameter names still work (backward compatible)
data <- simulate_onehot_data(
  actors_per_group = 5,
  actions = c("A", "B", "C"),
  seed = 42
)

## End(Not run)

```

simulate_prediction *Simulate Prediction/Regression Data with Known Coefficients*

Description

Generate a regression dataset with both continuous and categorical predictors, a non-linear term, and known ground-truth coefficients. A richer version of [simulate_regression](#) for testing prediction workflows.

Usage

```

simulate_prediction(
  n,
  coefs,
  cat_levels = NULL,
  cat_effects = NULL,
  error_sd = 1,
  predictor_means = NULL,
  predictor_sds = NULL,
  seed = NULL
)

```

Arguments

n	Integer. Sample size.
coefs	Named numeric vector. Must include names matching continuous predictors (e.g. x1, x2). May include "(Intercept)". Non-linear terms are NOT automatically generated; this specifies the linear part.
cat_levels	Named list. Each element is a character vector of factor levels for a categorical predictor. Default: NULL (no categorical predictors).
cat_effects	Named list of numeric vectors. Effect of each level (same length and names as cat_levels). Default: random effects.
error_sd	Positive numeric. Residual standard deviation. Default: 1.

predictor_means	Named numeric vector or NULL. Means for continuous predictors. Default: all zeros.
predictor_sds	Named numeric vector or NULL. SDs for continuous predictors. Default: all ones.
seed	Integer or NULL. Random seed.

Value

A `saqr_sim` object with:

`$data` data.frame with columns `y`, continuous predictors, and categorical predictors (factors).

`$params` list with `coefs`, `cat_effects`, `error_sd`, `predictor_means`, `predictor_sds`, `r_squared` (population R^2).

Examples

```
r <- simulate_prediction(
  n = 200,
  coefs = c("(Intercept)" = 5, x1 = 2, x2 = -1),
  cat_levels = list(treatment = c("control", "drug_a", "drug_b")),
  cat_effects = list(treatment = c(0, 3, 5)),
  error_sd = 2, seed = 42
)
summary(lm(y ~ ., data = r$data))
r$params$r_squared
```

<code>simulate_regression</code>	<i>Simulate Linear Regression Data with Known Coefficients</i>
----------------------------------	--

Description

Generate a dataset suitable for linear regression with fully specified ground-truth coefficients and predictor standard deviations. Designed so that `lm(y ~ ., data = r$data)` recovers the true coefficients at large `n`.

Usage

```
simulate_regression(coefs, predictor_sds, error_sd, n, seed = NULL)
```

Arguments

<code>coefs</code>	Named numeric vector. May include <code>"(Intercept)"</code> . All other names become predictor column names. Predictors are generated as independent $N(0, \sigma_j)$ variables.
<code>predictor_sds</code>	Named numeric vector. Names must exactly match the non-intercept names in <code>coefs</code> . All values must be positive.

error_sd	Positive numeric. Standard deviation of the residuals.
n	Positive integer. Sample size.
seed	Integer or NULL. Random seed.

Value

A named list with elements:

data data.frame with column y and one column per predictor.

params list with coefs, predictor_sds, and error_sd.

Examples

```
coefs <- c("(Intercept)" = 2, x1 = 3, x2 = -1)
r <- simulate_regression(coefs = coefs,
                        predictor_sds = c(x1 = 1, x2 = 1),
                        error_sd = 0.5, n = 500, seed = 42)
coef(lm(y ~ ., data = r$data)) # should be close to c(2, 3, -1)
```

simulate_seq_clusters *Simulate Sequence Data with Known Cluster Structure*

Description

Generate wide-format sequence data where each row is drawn from one of K Markov chains (clusters), each governed by its own transition matrix. Supports explicit matrix supply or automatic random generation.

Usage

```
simulate_seq_clusters(
  trans_list = NULL,
  props = NULL,
  n = 300L,
  seq_length = 20L,
  init_probs = NULL,
  n_clusters = 3L,
  n_states = 10L,
  states = NULL,
  seed = NULL
)
```

Arguments

<code>trans_list</code>	Named or unnamed list of square numeric transition matrices, or NULL for automatic generation. When NULL, <code>n_clusters</code> and <code>n_states</code> are used to build random row-stochastic matrices.
<code>props</code>	Numeric vector of mixing proportions (need not sum to 1; normalised internally). Length must equal K. Defaults to equal mixing.
<code>n</code>	Positive integer. Total number of sequences to generate.
<code>seq_length</code>	Positive integer. Number of time-points per sequence (columns T1...T{seq_length}). Default 20.
<code>init_probs</code>	Either a numeric vector of initial state probabilities shared across clusters, a list of per-cluster vectors, or NULL (uniform). Must be named consistently with the state names derived from the transition matrix row/column names.
<code>n_clusters</code>	Positive integer. Number of clusters when <code>trans_list</code> = NULL. Default 3.
<code>n_states</code>	Positive integer. Number of states when <code>trans_list</code> = NULL. Default 10.
<code>states</code>	Character vector of state names when <code>trans_list</code> = NULL. Defaults to <code>paste0("S", seq_len(n_states))</code> .
<code>seed</code>	Integer or NULL. Random seed for reproducibility.

Value

A named list with elements:

`data` `data.frame` with columns T1...T{seq_length} (character state labels) and `true_cluster` (integer 1...K).

`params` list with `trans_list` (the K matrices used), `props` (normalised mixing proportions), and `init_probs` (per-cluster initial probability vectors as a list).

Examples

```
m1 <- matrix(c(0.8, 0.2, 0.3, 0.7), nrow = 2, byrow = TRUE,
             dimnames = list(c("A","B"), c("A","B")))
m2 <- matrix(c(0.2, 0.8, 0.7, 0.3), nrow = 2, byrow = TRUE,
             dimnames = list(c("A","B"), c("A","B")))
r <- simulate_seq_clusters(trans_list = list(m1, m2), n = 100, seed = 1)
head(r$data)
r$params$props
```

<code>simulate_sequences</code>	<i>Simulate Markov Chain Sequences (Basic)</i>
---------------------------------	--

Description

Generate sequences of states from a Markov chain. Can either use provided transition matrix and initial probabilities, or auto-generate random ones with optional learning state names.

Usage

```
simulate_sequences(
  n_sequences = 1000,
  seq_length = 20,
  n_states = 8,
  states = NULL,
  use_learning_states = TRUE,
  categories = "all",
  smart_select = TRUE,
  trans_matrix = NULL,
  init_probs = NULL,
  na_range = c(0, 0),
  include_na = TRUE,
  include_params = FALSE,
  seed = NULL,
  num_rows = NULL,
  max_seq_length = NULL,
  state_names = NULL,
  learning_categories = NULL,
  transition_matrix = NULL,
  initial_probabilities = NULL,
  min_na = NULL,
  max_na = NULL,
  return_params = NULL
)
```

Arguments

<code>n_sequences</code>	Integer. Number of sequences (rows) to generate. Default: 1000.
<code>seq_length</code>	Integer. Maximum length of each sequence. Default: 20.
<code>n_states</code>	Integer. Number of states when auto-generating probabilities. Ignored if <code>trans_matrix</code> is provided. Default: 5.
<code>states</code>	Character vector. Names for states when auto-generating. If <code>NULL</code> , uses letters (A, B, C, ...) or learning states if enabled. Ignored if <code>trans_matrix</code> is provided. Default: <code>NULL</code> .
<code>use_learning_states</code>	Logical. If <code>TRUE</code> and auto-generating, uses realistic learning action verbs as state names. Default: <code>TRUE</code> .
<code>categories</code>	Character vector. Categories of learning states to use. Options: "metacognitive", "cognitive", "behavioral", "social", "motivational", "affective", "group_regulation", or "all". Only used if <code>use_learning_states = TRUE</code> . Default: "all".
<code>smart_select</code>	Logical. If <code>TRUE</code> , intelligently selects learning states based on <code>n_states</code> . Only used if <code>use_learning_states = TRUE</code> . Default: <code>TRUE</code> .
<code>trans_matrix</code>	Square numeric matrix of transition probabilities. Rows must sum to 1. Row names define state names. If <code>NULL</code> , random probabilities are generated based on <code>n_states</code> . Default: <code>NULL</code> .

init_probs	Named numeric vector of initial state probabilities. Must sum to 1. If NULL, random probabilities are generated. Default: NULL.
na_range	Integer vector of length 2 (min, max) or single integer (min=max). Range of NA values per sequence. Default: c(0, 0).
include_na	Logical. Whether to include NAs in sequences. Default: TRUE.
include_params	Logical. If TRUE, returns a list with sequences and the generating parameters. Default: FALSE.
seed	Integer or NULL. Random seed for reproducibility. Default: NULL.
num_rows	Deprecated. Use n_sequences instead.
max_seq_length	Deprecated. Use seq_length instead.
state_names	Deprecated. Use states instead.
learning_categories	Deprecated. Use categories instead.
transition_matrix	Deprecated. Use trans_matrix instead.
initial_probabilities	Deprecated. Use init_probs instead.
min_na	Deprecated. Use na_range instead.
max_na	Deprecated. Use na_range instead.
return_params	Deprecated. Use include_params instead.

Details

The function generates sequences by:

1. Sampling an initial state from `init_probs`.
2. For each subsequent position, sampling the next state based on the current state's row in `trans_matrix`.
3. If `include_na` is TRUE, trailing NAs are added to achieve a random number of NAs within `na_range`.

Auto-generation mode: When `trans_matrix` is NULL, the function automatically generates random transition probabilities and initial probabilities using `seqHMM::simulate_transition_probs()` and `seqHMM::simulate_initial_probs()`.

Learning States: When `use_learning_states` = TRUE, state names are drawn from a curated collection of 180+ student learning action verbs:

- **metacognitive:** Plan, Monitor, Evaluate, Reflect, ...
- **cognitive:** Read, Study, Analyze, Summarize, ...
- **behavioral:** Practice, Annotate, Research, Review, ...
- **social:** Collaborate, Discuss, Explain, Share, ...
- **motivational:** Focus, Persist, Explore, Strive, ...
- **affective:** Enjoy, Appreciate, Cope, Curious, ...
- **group_regulation:** Adapt, Cohesion, Consensus, ...

Value

If `include_params = FALSE` (default), a data frame with `n_sequences` rows and `seq_length` columns. Each row is a sequence of state names, potentially with trailing NAs.

If `include_params = TRUE`, a list containing:

- `sequences`: The data frame of sequences.
- `transition_matrix`: The transition probability matrix used.
- `initial_probabilities`: The initial probabilities used.
- `state_names`: The state names used.

See Also

[simulate_sequences_advanced](#) for sequences with stability modes, [simulate_long_data](#) for long-format educational data, [get_learning_states](#) for available learning state verbs, [simulate_tna_networks](#) for generating complete TNA networks.

Examples

```
## Not run:
# Simplest usage: all defaults (1000 sequences, 20 length, 5 states, learning states)
sequences <- simulate_sequences(seed = 42)

# Explicit new parameter names
sequences <- simulate_sequences(
  n_sequences = 100,
  seq_length = 15,
  n_states = 6,
  use_learning_states = TRUE,
  categories = c("metacognitive", "cognitive"),
  seed = 42
)

# Method 1: Provide your own transition matrix
trans_mat <- matrix(c(
  0.7, 0.2, 0.1,
  0.3, 0.5, 0.2,
  0.2, 0.3, 0.5
), nrow = 3, byrow = TRUE)
rownames(trans_mat) <- colnames(trans_mat) <- c("A", "B", "C")
init_probs <- c(A = 0.5, B = 0.3, C = 0.2)

sequences <- simulate_sequences(
  trans_matrix = trans_mat,
  init_probs = init_probs,
  seq_length = 20,
  n_sequences = 100
)

# Method 2: Auto-generate with letter names
sequences <- simulate_sequences(
```

```

    seq_length = 20,
    n_sequences = 100,
    n_states = 5,
    use_learning_states = FALSE,
    seed = 42
)

# Method 3: Auto-generate with learning state names
sequences <- simulate_sequences(
  seq_length = 25,
  n_sequences = 150,
  n_states = 6,
  use_learning_states = TRUE,
  categories = c("metacognitive", "cognitive"),
  seed = 123
)

# Method 4: Get sequences AND the generating parameters
result <- simulate_sequences(
  seq_length = 20,
  n_sequences = 100,
  n_states = 4,
  use_learning_states = TRUE,
  include_params = TRUE,
  seed = 42
)
result$sequences          # The sequences
result$transition_matrix  # The random transition matrix
result$state_names        # e.g., c("Plan", "Monitor", "Read", "Practice")

# Method 5: Custom state names
sequences <- simulate_sequences(
  seq_length = 20,
  n_sequences = 100,
  n_states = 4,
  states = c("Explore", "Learn", "Practice", "Master"),
  seed = 42
)

# Old parameter names still work (backward compatible)
sequences <- simulate_sequences(
  num_rows = 100,
  max_seq_length = 15,
  state_names = c("A", "B", "C"),
  seed = 42
)

## End(Not run)

```

simulate_sequences_advanced

Simulate Markov Chain Sequences (Advanced)

Description

Generate sequences of states from a Markov chain with advanced stability and instability modes. Supports stable transitions, probability perturbation, and unlikely jump modes.

Usage

```
simulate_sequences_advanced(
  n_sequences = 1000,
  seq_length = 20,
  n_states = 8,
  states = NULL,
  use_learning_states = TRUE,
  categories = "all",
  trans_matrix = NULL,
  init_probs = NULL,
  stable_transitions = NULL,
  stability_prob = 0.95,
  unstable_mode = "unlikely_jump",
  unstable_random_transition_prob = 0.4,
  unstable_perturb_noise = 0.5,
  unlikely_prob_threshold = 0.1,
  na_range = c(0, 0),
  include_na = TRUE,
  seed = NULL,
  transition_matrix = NULL,
  initial_probabilities = NULL,
  num_rows = NULL,
  max_seq_length = NULL,
  min_na = NULL,
  max_na = NULL
)
```

Arguments

n_sequences	Integer. Number of sequences (rows) to generate. Default: 1000.
seq_length	Integer. Maximum length of each sequence. Default: 20.
n_states	Integer. Number of states when auto-generating probabilities. Ignored if trans_matrix is provided. Default: 5.
states	Character vector. Names for states when auto-generating. If NULL, uses letters (A, B, C, ...) or learning states if enabled. Ignored if trans_matrix is provided. Default: NULL.

use_learning_states	Logical. If TRUE and auto-generating, uses realistic learning action verbs as state names. Default: TRUE.
categories	Character vector. Categories of learning states to use. Only used if use_learning_states = TRUE. Default: "all".
trans_matrix	Square numeric matrix of transition probabilities. Rows must sum to 1. Row names define state names. If NULL, random probabilities are generated. Default: NULL.
init_probs	Named numeric vector of initial state probabilities. Must sum to 1. If NULL, random probabilities are generated. Default: NULL.
stable_transitions	List of character vectors. Each vector contains two state names defining a stable transition pair (from, to). Default: NULL (no stable transitions).
stability_prob	Numeric in (0 to 1). Probability of following a stable transition when in a stable state. Default: 0.95.
unstable_mode	Character. Mode for unstable transitions. One of: • "random_jump": Uniform random jump to any state. • "perturb_prob": Perturb transition probabilities with noise. • "unlikely_jump": Jump to states with low probability. Default: "unlikely_jump".
unstable_random_transition_prob	Numeric in (0 to 1). Probability of taking an unstable action. Default: 0.4.
unstable_perturb_noise	Numeric in (0 to 1). Noise factor for probability perturbation mode. Default: 0.5.
unlikely_prob_threshold	Numeric in (0 to 1). Threshold below which transitions are considered "unlikely". Default: 0.1.
na_range	Integer vector of length 2 (min, max) or single integer (min=max). Range of NA values per sequence. Default: c(0, 0).
include_na	Logical. Whether to include NAs in sequences. Default: TRUE.
seed	Integer or NULL. Random seed for reproducibility. Default: NULL.
transition_matrix	Deprecated. Use trans_matrix instead.
initial_probabilities	Deprecated. Use init_probs instead.
num_rows	Deprecated. Use n_sequences instead.
max_seq_length	Deprecated. Use seq_length instead.
min_na	Deprecated. Use na_range instead.
max_na	Deprecated. Use na_range instead.

Details

The function extends basic Markov chain simulation with:

Stable Transitions: If the current state has a defined stable transition and a random draw is below `stability_prob`, the sequence follows the stable transition directly.

Unstable Modes (when not following stable transitions):

- "random_jump": With probability `unstable_random_transition_prob`, jump uniformly to any state.
- "perturb_prob": Multiply transition probabilities by random noise in range `[1-noise, 1+noise]`, then renormalize.
- "unlikely_jump": With probability `unstable_random_transition_prob`, jump to a state with transition probability below threshold.

NAs are added at the end of sequences, preserving at least 2 non-NA values.

Value

A data frame with `n_sequences` rows and `seq_length` columns. Each row is a sequence of state names, potentially with trailing NAs.

Examples

```
## Not run:
# Simplest usage: all defaults
sequences <- simulate_sequences_advanced(seed = 42)

# Create a 4-state transition matrix
trans_mat <- matrix(c(
  0.6, 0.2, 0.1, 0.1,
  0.1, 0.7, 0.1, 0.1,
  0.1, 0.1, 0.6, 0.2,
  0.2, 0.1, 0.1, 0.6
), nrow = 4, byrow = TRUE)
rownames(trans_mat) <- colnames(trans_mat) <- c("A", "B", "C", "D")

init_probs <- c(A = 0.25, B = 0.25, C = 0.25, D = 0.25)

# Define stable transitions: A->B and C->D are "stable"
stable <- list(c("A", "B"), c("C", "D"))

# Generate sequences with unlikely_jump mode
sequences <- simulate_sequences_advanced(
  trans_matrix = trans_mat,
  init_probs = init_probs,
  seq_length = 30,
  n_sequences = 100,
  stable_transitions = stable,
  stability_prob = 0.95,
  unstable_mode = "unlikely_jump",
  unstable_random_transition_prob = 0.3,
```

```

    na_range = c(0, 5)
  )

  # Old parameter names still work (backward compatible)
  sequences <- simulate_sequences_advanced(
    transition_matrix = trans_mat,
    initial_probabilities = init_probs,
    max_seq_length = 30,
    num_rows = 100
  )

  ## End(Not run)

```

simulate_survival	<i>Simulate Survival Data (Cox Proportional Hazards)</i>
-------------------	--

Description

Generate right-censored survival data from a Cox proportional-hazards generative model. The hazard for subject i is

$$h(t \mid X_i) = h_0(t) \exp(X_i \beta),$$

where $h_0(t)$ is a parametric baseline hazard (Weibull, exponential, or Gompertz). Event times are drawn by inverse-CDF sampling given each subject's linear predictor, then administrative/exponential censoring is applied so that roughly `censoring_rate` of observations are censored. Designed so that `coxph(Surv(time, status) ~ ., data)` recovers betas at large n .

Usage

```

simulate_survival(
  n = 300,
  n_covariates = 2,
  betas = NULL,
  baseline = c("weibull", "exponential", "gompertz"),
  lambda = 0.1,
  shape = 1,
  censoring_rate = 0.3,
  covariate_type = c("normal", "binary"),
  seed = NULL
)

```

Arguments

<code>n</code>	Integer. Number of subjects. Default: 300.
<code>n_covariates</code>	Integer. Number of covariates p . Default: 2. Ignored when <code>betas</code> is supplied (its length wins).

betas	Numeric vector or NULL. True log hazard ratios, one per covariate. When NULL, a moderate alternating sequence is auto-generated (e.g. $c(0.5, -0.3, 0.5, -0.3, \dots)$).
baseline	Character. Baseline hazard family: "weibull" (default), "exponential", or "gompertz".
lambda	Positive numeric. Baseline scale parameter. Default: 0.1.
shape	Positive numeric. Baseline shape parameter. For Weibull this is the Weibull shape; for Gompertz the level parameter; ignored for exponential. Default: 1.
censoring_rate	Numeric in $[0, 1)$. Target proportion of censored observations. Default: 0.3.
covariate_type	Character. Covariate distribution: "normal" (default, standard normal) or "binary" (Bernoulli(0.5), coded 0/1).
seed	Integer or NULL. Random seed.

Details

Inverse-CDF event times use the closed-form survival inverse for each baseline. With $U \sim \text{Unif}(0, 1)$ and linear predictor $\eta = X\beta$: Weibull $t = (-\log U / (\lambda e^\eta))^{1/\text{shape}}$; exponential $t = -\log U / (\lambda e^\eta)$; Gompertz $t = (1/\text{shape}) \log(1 - \text{shape} \log U / (\lambda e^\eta))$.

Value

A `saqr_sim` object with:

`$data` data.frame with columns `time` (numeric event or censoring time), `status` (1 = event, 0 = censored), and `X1...Xp` (covariates).

`$params` list with `betas` (true log hazard ratios), `baseline`, `lambda`, `shape`, `covariate_type`, and `realized_censoring_rate`.

Examples

```
r <- simulate_survival(n = 400, n_covariates = 2, seed = 1)
print(r)
r$params$betas
r$params$realized_censoring_rate
head(as.data.frame(r))

# Binary covariates, exponential baseline, explicit effects
r2 <- simulate_survival(n = 500, betas = c(0.8, -0.5),
                        baseline = "exponential",
                        covariate_type = "binary", seed = 42)

summary(r2)
```

`simulate_tna_datasets` *Simulate TNA Datasets (Sequences + Models + Probabilities)*

Description

Simulate complete TNA datasets including simulated sequences, fitted models, and generating probabilities. This function combines the full simulation workflow: probability generation -> sequence simulation -> model fitting.

Use [simulate_tna_networks](#) if you only need the fitted models.

Supports using realistic learning action verbs as state names for educational research simulations.

Usage

```
simulate_tna_datasets(  
  n_datasets = 1,  
  n_states = 8,  
  n_sequences = 100,  
  seq_length = 30,  
  states = NULL,  
  use_learning_states = TRUE,  
  categories = NULL,  
  smart_select = TRUE,  
  na_range = c(0, 0),  
  model_type = "tna",  
  use_advanced = FALSE,  
  stable_transitions = NULL,  
  stability_prob = 0.95,  
  unstable_mode = "random_jump",  
  unstable_random_transition_prob = 0.5,  
  include_data = TRUE,  
  include_probs = TRUE,  
  seed = NULL,  
  verbose = TRUE,  
  state_names = NULL,  
  learning_categories = NULL,  
  num_rows = NULL,  
  max_seq_length = NULL,  
  min_na = NULL,  
  max_na = NULL,  
  include_probabilities = NULL  
)  
  
generate_tna_datasets(...)  
  
generate_sequence_data(...)
```

Arguments

<code>n_datasets</code>	Integer. Number of datasets to generate. Default: 1.
<code>n_states</code>	Integer. Number of states in each network. Default: 5.
<code>n_sequences</code>	Integer. Number of sequences to simulate per network. Default: 100.
<code>seq_length</code>	Integer. Maximum length of each sequence. Default: 30.
<code>states</code>	Character vector. Names for the states. If NULL and <code>use_learning_states</code> = FALSE, uses letters (A, B, C, ...). Ignored if <code>use_learning_states</code> = TRUE. Default: NULL.
<code>use_learning_states</code>	Logical. If TRUE, uses learning action verbs as state names (e.g., "Plan", "Monitor", "Read"). Default: TRUE.
<code>categories</code>	Character vector. Which categories of learning verbs to use. Options: "metacognitive", "cognitive", "behavioral", "social", "motivational", "affective", "group_regulation", or "all". If NULL and <code>use_learning_states</code> = TRUE, a random category is selected. Only used if <code>use_learning_states</code> = TRUE. Default: NULL.
<code>smart_select</code>	Logical. If TRUE, intelligently selects learning states based on <code>n_states</code> (small networks use fewer categories). Only used if <code>use_learning_states</code> = TRUE. Default: TRUE.
<code>na_range</code>	Integer vector of length 2 (min, max) or single integer. Range of NA values per sequence. Default: c(0, 0).
<code>model_type</code>	Character. Type of TNA model to fit. One of: "tna", "ftna", "ctna", "atna". Default: "tna".
<code>use_advanced</code>	Logical. If TRUE, uses <code>simulate_sequences_advanced()</code> with stability modes. If FALSE, uses basic <code>simulate_sequences()</code> . Default: FALSE.
<code>stable_transitions</code>	List of character vectors defining stable transitions (only used if <code>use_advanced</code> = TRUE). Default: NULL.
<code>stability_prob</code>	Numeric (0 to 1). Probability of following stable transitions (only used if <code>use_advanced</code> = TRUE). Default: 0.95.
<code>unstable_mode</code>	Character. Mode for unstable transitions: "random_jump", "perturb_prob", or "unlikely_jump". (Only used if <code>use_advanced</code> = TRUE). Default: "random_jump".
<code>unstable_random_transition_prob</code>	Numeric (0 to 1). Probability of unstable action (only used if <code>use_advanced</code> = TRUE). Default: 0.5.
<code>include_data</code>	Logical. If TRUE, includes the generating sequence data in the output. Default: TRUE.
<code>include_probs</code>	Logical. If TRUE, includes the generating transition matrix and initial probabilities in the output. Default: TRUE.
<code>seed</code>	Integer or NULL. Random seed for reproducibility. If NULL, no seed is set. Default: NULL.
<code>verbose</code>	Logical. If TRUE, prints progress messages. Default: TRUE.
<code>state_names</code>	Deprecated. Use <code>states</code> instead.

learning_categories	Deprecated. Use categories instead.
num_rows	Deprecated. Use n_sequences instead.
max_seq_length	Deprecated. Use seq_length instead.
min_na	Deprecated. Use na_range instead.
max_na	Deprecated. Use na_range instead.
include_probabilities	Deprecated. Use include_probs instead.
...	Arguments passed to simulate_tna_datasets.

Details

The function generates sequence data by:

1. Generating random transition and initial probabilities using `seqHMM::simulate_transition_probs()` and `seqHMM::simulate_initial_probs()`.
2. Simulating sequences from those probabilities using either `simulate_sequences()` or `simulate_sequences_advance()`.
3. Fitting a TNA model to the sequences using `fit_network_model()`.

Learning States: When `use_learning_states = TRUE`, state names are drawn from a curated collection of 180+ student learning action verbs organized into 7 categories:

- **metacognitive:** Plan, Monitor, Evaluate, Reflect, Regulate, ...
- **cognitive:** Read, Study, Analyze, Summarize, Memorize, ...
- **behavioral:** Practice, Annotate, Research, Review, Write, ...
- **social:** Collaborate, Discuss, Explain, Share, Teach, ...
- **motivational:** Focus, Persist, Explore, Strive, Commit, ...
- **affective:** Enjoy, Appreciate, Cope, Manage, Curious, ...
- **group_regulation:** Adapt, Cohesion, Consensus, Coregulate, ...

Value

A list of length `n_datasets`. Each element is a list containing:

model The fitted TNA model object.

transition_probs The generating transition matrix (if `include_probs = TRUE`).

initial_probs The generating initial probabilities (if `include_probs = TRUE`).

sequences The simulated sequence data frame (if `include_data = TRUE`).

params List of parameters used for this network.

See Also

[simulate_tna_networks](#) for generating TNA network objects, [get_learning_states](#) for retrieving learning state verbs, [select_states](#) for intelligent state selection, [list_learning_categories](#) for viewing available categories, [simulate_sequences](#) for basic sequence simulation, [fit_network_model](#) for model fitting.

Examples

```
## Not run:
# Simplest usage - generates 1 dataset with sequences + model + probabilities
data <- simulate_tna_datasets(seed = 42)
data[[1]]$sequences      # The sequence data
data[[1]]$model          # The fitted TNA model
data[[1]]$transition_probs # The generating probabilities

# Generate 5 datasets with learning states
data <- simulate_tna_datasets(
  n_datasets = 5,
  n_states = 4,
  seed = 42
)

# Generate with specific learning category
learning_data <- simulate_tna_datasets(
  n_datasets = 5,
  n_states = 6,
  categories = c("metacognitive", "cognitive"),
  seed = 42
)

# View the state names
learning_data[[1]]$params$state_names
# e.g., c("Plan", "Monitor", "Read", "Practice", "Discuss", "Focus")

# Generate with letter names (disable learning states)
letter_data <- simulate_tna_datasets(
  n_datasets = 5,
  n_states = 8,
  use_learning_states = FALSE,
  seed = 123
)

## End(Not run)
```

simulate_tna_matrix	<i>Simulate TNA Transition Matrix with Node Groupings</i>
---------------------	---

Description

Simulate a transition matrix with node groupings, compatible with `tna::plot_htna()` (hierarchical) and `tna::plot_mlna()` (multilevel) visualizations. By default creates a 25-node matrix (5 nodes x 5 types) using learning category names.

This is a convenience wrapper around [simulate_htna](#).

Usage

```
simulate_tna_matrix(
  nodes_per_group = 5,
  group_names = c("Metacognitive", "Cognitive", "Behavioral", "Social", "Motivational"),
  n_groups = 5,
  edge_prob_range = c(0, 1),
  self_loops = FALSE,
  use_learning_states = TRUE,
  categories = c("metacognitive", "cognitive", "behavioral", "social", "motivational"),
  within_prob = 0.4,
  between_prob = 0.15,
  node_prefix = "N",
  seed = NULL,
  verbose = TRUE,
  learning_categories = NULL
)

generate_tna_matrix(...)
```

Arguments

<code>nodes_per_group</code>	Integer. Number of nodes per group. Default: 5.
<code>group_names</code>	Character vector. Names for each group. Default uses learning categories: "Metacognitive", "Cognitive", "Behavioral", "Social", "Motivational".
<code>n_groups</code>	Integer. Number of groups. Default: 5.
<code>edge_prob_range</code>	Numeric vector of length 2. Range for edge weights <code>c(min, max)</code> . Default: <code>c(0, 1)</code> .
<code>self_loops</code>	Logical. Allow self-loops (diagonal elements). Default: FALSE.
<code>use_learning_states</code>	Logical. Use learning state verbs as node names. Default: TRUE.
<code>categories</code>	Character vector. Categories for node names, one per group. Default: <code>c("metacognitive", "cognitive", "behavioral", "social", "motivational")</code> .
<code>within_prob</code>	Numeric. Probability of edges within each group. Default: 0.4.
<code>between_prob</code>	Numeric. Probability of edges between groups. Default: 0.15.
<code>node_prefix</code>	Character. Prefix for node names when not using learning states. Default: "N".
<code>seed</code>	Integer or NULL. Random seed. Default: NULL.
<code>verbose</code>	Logical. Print progress messages. Default: TRUE.
<code>learning_categories</code>	Deprecated. Use <code>categories</code> instead.
<code>...</code>	Arguments passed to <code>simulate_tna_matrix</code> .

Details

This function generates a random transition matrix and node groupings. The output can be used with both hierarchical and multilevel TNA plots:

- For `plot_htna()`: use `net$node_types` directly
- For `plot_mlna()`: use `layers = net$node_types`

Value

A list with two elements:

matrix Square transition matrix (rows sum to 1) with named rows/columns.

node_types Named list mapping group names to node names. Use as `node_types` for `plot_htna()` or as `layers` for `plot_mlna()`.

See Also

[simulate_htna](#) for the underlying function, [simulate_matrix](#) for basic matrix simulation, [simulate_tna_networks](#) for TNA model objects.

Examples

```
## Not run:
# Default: 5 groups x 5 nodes = 25 node matrix
net <- simulate_tna_matrix(seed = 42)
net$matrix
net$node_types # Metacognitive, Cognitive, Behavioral, Social, Motivational

# Use with plot_htna
plot_htna(net$matrix, net$node_types, layout = "polygon")

# Use with plot_mlna
plot_mlna(net$matrix, layers = net$node_types)

# Custom group names (3 groups)
net <- simulate_tna_matrix(
  nodes_per_group = 6,
  group_names = c("Macro", "Meso", "Micro"),
  seed = 42
)

# Custom categories per group
net <- simulate_tna_matrix(
  nodes_per_group = 4,
  group_names = c("Teacher", "Student", "System"),
  categories = c("metacognitive", "cognitive", "behavioral"),
  seed = 123
)

## End(Not run)
```

simulate_tna_network	<i>Simulate a Single TNA Network</i>
----------------------	--------------------------------------

Description

Generate a single fitted TNA network model with learning state names. This is the simplest way to get a ready-to-use tna model object.

Usage

```
simulate_tna_network(  
  n_states = 9,  
  n_sequences = NULL,  
  seq_length = 25,  
  categories = c("metacognitive", "cognitive"),  
  seed = NULL  
)
```

Arguments

n_states	Integer. Number of states (8-10 recommended). Default: 9.
n_sequences	Integer or NULL. Number of sequences to simulate. If NULL (default), randomly selects between 600-1200.
seq_length	Integer. Length of each sequence. Default: 25.
categories	Character vector. Learning state categories to use. Options: "metacognitive", "cognitive", "behavioral", "social", "motivational", "affective", "group_regulation", "lms", or "all". Default: c("metacognitive", "cognitive").
seed	Integer or NULL. Random seed for reproducibility. Default: NULL.

Details

This function provides the simplest interface for generating a TNA network:

1. Selects learning state names from specified categories
2. Generates random transition probabilities
3. Simulates Markov chain sequences
4. Fits and returns a tna model

The returned model is 100% compatible with all tna package functions.

Value

A tna model object (class "tna") ready for use with all tna functions: plot(), centralities(), communities(), etc.

See Also

[simulate_tna_networks](#) for generating multiple networks, [simulate_sequences](#) for generating sequences without fitting, [fit_network_model](#) for fitting models to existing data.

Examples

```
library(SaqrLab)

# Generate a network (simplest usage)
model <- simulate_tna_network(seed = 42)

# Use with tna functions
## Not run:
library(tna)
plot(model)
centralities(model)
communities(model)

## End(Not run)

# Custom configuration
model <- simulate_tna_network(
  n_states = 8,
  n_sequences = 300,
  categories = "group_regulation",
  seed = 123
)

# Access model components
model$weights
model$initial_probs
```

simulate_tna_networks *Simulate TNA Network Objects*

Description

Simulate multiple TNA network objects (fitted models) for simulation studies. This function creates TNA model objects by simulating sequence data and fitting models. For generating raw sequence data with associated probabilities, use [simulate_tna_datasets](#). For group TNA models, use [simulate_group_tna_networks](#).

Usage

```
simulate_tna_networks(
  n_networks = 1,
  n_states = 8,
  n_sequences = 100,
```

```

    seq_length = 30,
    model_type = "tna",
    use_learning_states = TRUE,
    categories = NULL,
    seed = NULL,
    verbose = TRUE,
    num_sequences = NULL,
    max_seq_length = NULL,
    learning_categories = NULL,
    ...
)

generate_tna_networks(...)

```

Arguments

n_networks	Integer. Number of networks to generate. Default: 1.
n_states	Integer. Number of states/actions in each network. Default: 5.
n_sequences	Integer. Number of sequences to simulate per network. Default: 100.
seq_length	Integer. Maximum length of each sequence. Default: 30.
model_type	Character. Type of TNA model to fit: "tna", "ftna", "ctna", "atna". Default: "tna".
use_learning_states	Logical. If TRUE, uses learning action verbs as state names. Default: TRUE.
categories	Character vector or NULL. Which categories of learning verbs to use. Options: "metacognitive", "cognitive", "behavioral", "social", "motivational", "affective", "group_regulation", or "all". If NULL (default), randomly selects one category.
seed	Integer or NULL. Random seed for reproducibility. Default: NULL.
verbose	Logical. If TRUE, prints progress messages. Default: TRUE.
num_sequences	Deprecated. Use n_sequences instead.
max_seq_length	Deprecated. Use seq_length instead.
learning_categories	Deprecated. Use categories instead.
...	Arguments passed to simulate_tna_networks.

Details

This function differs from [simulate_tna_datasets](#) in that it returns only the fitted model objects, not the underlying sequence data or generating probabilities. Use this when you need TNA network objects for simulation studies or method comparisons.

Random Category Selection: When `categories = NULL` and `use_learning_states = TRUE`, a random learning category is selected. Available categories: metacognitive, cognitive, behavioral, social, motivational, affective, group_regulation.

Value

A list of length `n_networks` containing fitted TNA model objects. Each element is a tna model object (class "tna").

See Also

[simulate_group_tna_networks](#) for group TNA models, [simulate_tna_datasets](#) for generating complete datasets with probabilities, [fit_network_model](#) for model fitting, [get_learning_states](#) for learning state verbs.

Examples

```
## Not run:
# Generate 5 TNA networks with random learning category
nets <- simulate_tna_networks(n_networks = 5, seed = 42)

# Generate networks with specific category
meta_nets <- simulate_tna_networks(
  n_networks = 3,
  n_states = 6,
  categories = "metacognitive",
  seed = 123
)

# Generate filtered TNA networks
ftna_nets <- simulate_tna_networks(
  n_networks = 5,
  model_type = "ftna",
  seed = 456
)

# Use the networks
plot(nets[[1]])
tna::centralities(nets[[1]])

# Old parameter names still work
nets <- simulate_tna_networks(
  n_networks = 3,
  num_sequences = 50,
  max_seq_length = 25,
  seed = 42
)

## End(Not run)
```

Description

Generate data for a two-group comparison with fully specified ground-truth means and standard deviations. Designed so that `t.test(score ~ group, data = r$data)` recovers the true difference at large `n`.

Usage

```
simulate_ttest(  
  n_a,  
  n_b,  
  mean_a,  
  mean_b,  
  sd_a = 1,  
  sd_b = sd_a,  
  labels = c("A", "B"),  
  seed = NULL  
)
```

Arguments

<code>n_a</code>	Integer. Sample size for group A.
<code>n_b</code>	Integer. Sample size for group B.
<code>mean_a</code>	Numeric. Population mean for group A.
<code>mean_b</code>	Numeric. Population mean for group B.
<code>sd_a</code>	Positive numeric. Standard deviation for group A. Default: 1.
<code>sd_b</code>	Positive numeric. Standard deviation for group B. Default: same as <code>sd_a</code> .
<code>labels</code>	Character vector of length 2. Group labels. Default: <code>c("A", "B")</code> .
<code>seed</code>	Integer or NULL. Random seed.

Value

A `saqr_sim` object with:

`$data` data.frame with columns `group` (factor) and `score` (numeric).

`$params` list with `mean_a`, `mean_b`, `sd_a`, `sd_b`, `n_a`, `n_b`, `cohens_d` (true Cohen's `d` based on pooled SD).

Examples

```
r <- simulate_ttest(n_a = 50, n_b = 50, mean_a = 100, mean_b = 105, seed = 1)  
t.test(score ~ group, data = r$data)  
r$params$cohens_d
```

summarize_grid_results

Summarize Grid Simulation Results

Description

Summarize results from `run_grid_simulation()`, filtering by parameter ranges and computing aggregated performance metrics. Provides detailed summaries at both the setting level and across all selected settings.

Usage

```
summarize_grid_results(
  grid_results_list,
  n_sequences_range = NULL,
  seq_length_range = NULL,
  min_na_range = NULL,
  max_na_range = NULL,
  level_context = 0.05,
  print_output = TRUE,
  print_aggregated_overall = TRUE,
  print_aggregated_edges = TRUE,
  print_settings_summary = TRUE,
  num_rows_range = NULL,
  max_seq_length_range = NULL
)

analyze_grid_results(...)
```

Arguments

<code>grid_results_list</code>	List output from <code>run_grid_simulation()</code> .
<code>n_sequences_range</code>	Numeric vector of length 2 (min, max) or NULL. Filter settings by <code>n_sequences</code> .
<code>seq_length_range</code>	Numeric vector of length 2 (min, max) or NULL. Filter settings by <code>seq_length</code> .
<code>min_na_range</code>	Numeric vector of length 2 (min, max) or NULL. Filter settings by <code>min_na</code> .
<code>max_na_range</code>	Numeric vector of length 2 (min, max) or NULL. Filter settings by <code>max_na</code> .
<code>level_context</code>	Numeric. Significance level used for calculations (for context in output). Default: 0.05.
<code>print_output</code>	Logical. Master switch for console printing. Default: TRUE.
<code>print_aggregated_overall</code>	Logical. Print averaged overall performance metrics across selected settings. Default: TRUE.

```

print_aggregated_edges
    Logical. Print aggregated edge significance summary. Default: TRUE.
print_settings_summary
    Logical. Print summary table for each selected setting. Default: TRUE.
num_rows_range
    Deprecated. Use n_sequences_range instead.
max_seq_length_range
    Deprecated. Use seq_length_range instead.
...
    Arguments passed to summarize_grid_results.

```

Details

The function performs comprehensive analysis:

Filtering: Selects settings where all parameters fall within specified ranges.

Aggregation from Input Summaries: Extracts and averages overall performance and edge significance from the original aggregated_summary in each setting.

Run-Level Metrics: Recomputes TP/TN/FP/FN counts from per-edge data, then calculates Sensitivity, Specificity, FPR, FNR, Accuracy, and MCC per run.

Setting-Level Metrics: Two approaches:

1. Mean/Median/SD of run-level metrics.
2. Metrics computed from total counts across all runs (more robust).

The selected_settings_summary_df uses approach #2 for the most accurate overall metrics per setting.

Value

A list containing:

n_selected Number of settings matching the filter criteria.

aggregated_summary List with:

- overall_performance: Averaged metrics across settings.
- edge_significance: Aggregated edge-level statistics.

selected_settings_summary_df Data frame with per-setting metrics calculated from total TP/TN/FP/FN counts.

compiled_individual_runs List with detailed run-level data:

- all_raw_summaries: Combined bootstrap summaries.
- all_per_edge_performance: Combined per-edge results.
- run_level_performance_metrics: Metrics per run.
- setting_level_summary_stats: Mean/Median/SD of run metrics.

Returns NULL (invisibly) if no settings match.

Examples

```
## Not run:
# After running grid simulation
grid_results <- run_grid_simulation(...)

# Analyze all results
analysis <- summarize_grid_results(grid_results)

# Filter to specific parameter ranges
analysis_filtered <- summarize_grid_results(
  grid_results,
  n_sequences_range = c(100, 200),
  seq_length_range = c(30, 50),
  max_na_range = c(0, 5)
)

# Access the detailed run-level metrics
run_metrics <- analysis$compiled_individual_runs$run_level_performance_metrics

# Access setting-level summary
setting_summary <- analysis$selected_settings_summary_df

# Old parameter names still work
analysis_filtered <- summarize_grid_results(
  grid_results,
  num_rows_range = c(100, 200),
  max_seq_length_range = c(30, 50)
)

## End(Not run)
```

summarize_networks	<i>Summarize Multiple Networks</i>
--------------------	------------------------------------

Description

Compute aggregate statistics across multiple TNA network models.

Usage

```
summarize_networks(
  model_list,
  include = c("density", "centrality", "edges"),
  threshold = 0.01,
  centrality_measures = c("OutStrength", "InStrength")
)
```

Arguments

<code>model_list</code>	A list of TNA model objects.
<code>include</code>	Character vector. What to include in the summary. Options: "density" (network density), "centrality" (centrality measures), "edges" (edge weight statistics), or "all" for everything. Default: <code>c("density", "centrality", "edges")</code> .
<code>threshold</code>	Numeric. Minimum edge weight to consider present for density calculation. Default: 0.01.
<code>centrality_measures</code>	Character vector. Which centrality measures to summarize. Default: <code>c("OutStrength", "InStrength")</code> .

Details

This function is useful for summarizing results from simulation studies where many networks are generated. It provides both per-network and aggregate statistics.

Value

A list containing:

- `summary_table`: Data frame with per-network statistics.
- `aggregate`: Named list of aggregate statistics across all networks.
- `n_networks`: Number of networks summarized.

See Also

[summarize_simulation\(\)](#) for simulation result summaries, [batch_fit_models\(\)](#) for fitting multiple models.

Examples

```
## Not run:
# Generate and fit multiple networks
datasets <- lapply(1:20, function(i) {
  simulate_sequences(trans_mat, init_probs, 20, 100)
})
models <- batch_fit_models(datasets)

# Summarize networks
network_summary <- summarize_networks(models)
print(network_summary$aggregate)

## End(Not run)
```

summarize_simulation *Summarize Simulation Results*

Description

Compute summary statistics for simulation results, optionally grouped by parameter combinations or other factors.

Usage

```
summarize_simulation(
  results,
  by = NULL,
  metrics = c("mean", "sd", "ci"),
  value_cols = NULL,
  na.rm = TRUE
)
```

Arguments

<code>results</code>	A data frame of simulation results, typically from <code>run_network_simulation()</code> or <code>run_bootstrap_simulation()</code> .
<code>by</code>	Character vector. Column names to group by when computing summaries. Default: NULL (compute overall summary).
<code>metrics</code>	Character vector. Which summary metrics to compute. Options: "mean" (arithmetic mean), "sd" (standard deviation), "median" (median value), "ci" (95 percent confidence interval), "min" (minimum), "max" (maximum), "n" (count), or "all" to compute all metrics. Default: <code>c("mean", "sd", "ci")</code> .
<code>value_cols</code>	Character vector. Names of columns containing numeric values to summarize. If NULL, auto-detects numeric columns. Default: NULL.
<code>na.rm</code>	Logical. Whether to remove NA values when computing statistics. Default: TRUE.

Details

This function provides flexible summarization of simulation results. It automatically detects numeric columns and computes requested statistics.

Confidence intervals are computed as $\text{mean} \pm 1.96 * \text{SE}$, where $\text{SE} = \text{sd} / \sqrt{n}$.

Value

A data frame with summary statistics. If `by` is specified, contains one row per unique combination of grouping variables.

See Also

[summarize_networks\(\)](#) for summarizing network metrics, [run_network_simulation\(\)](#) for running simulations.

Examples

```
## Not run:
# Run a simulation
results <- run_network_simulation(trans_mat, init_probs,
                                num_rows_values = c(50, 100, 200),
                                n_runs = 20)

# Overall summary
summarize_simulation(results)

# Summary by sample size
summarize_simulation(results, by = "num_rows")

# Summary with all metrics
summarize_simulation(results, by = "num_rows", metrics = "all")

## End(Not run)
```

tidy_simulation_results

Flatten Simulation Results into One Tidy Data Frame

Description

Stack the `$data` of several [saqr_sim](#) objects into a single tidy data.frame. Each sim contributes its rows plus a `.case` column (the list name) and one column per scalar (length-1) ground-truth parameter in `$params`. Non-scalar parameters (matrices, vectors) are skipped. Columns are unioned across cases, so cases with different parameter sets still bind safely.

Usage

```
tidy_simulation_results(x)
```

Arguments

`x` A named list of [saqr_sim](#) objects (for example the output of [run_scenario](#)), or a single `saqr_sim` object.

Value

A single base data.frame with all data rows stacked, a `.case` column, and one column per scalar parameter.

Examples

```
s <- run_scenario("power_ttest", seed = 1)
tidy_simulation_results(s)
```

validate_recovery	<i>Score Parameter Recovery Against Simulated Ground Truth</i>
-------------------	--

Description

Compare a method's point estimates to the known true values that generated a simulation, matched **by name**. This is the core "did my method recover the truth I simulated?" check of the simulation laboratory: feed in a [saqr_sim](#) object (or a plain named list / vector of true values) and a named vector of estimates, and get back a tidy one-row-per-parameter table of errors and tolerance flags.

Usage

```
validate_recovery(
  sim,
  estimates,
  params = NULL,
  tolerance = 0.1,
  relative = TRUE
)

## S3 method for class 'recovery_result'
print(x, ...)

## S3 method for class 'recovery_result'
summary(object, ...)
```

Arguments

sim	A saqr_sim object (its \$params supplies the ground truth) <i>or</i> a plain named list / named numeric vector of true values.
estimates	A named numeric vector (or named list of scalars) of point estimates produced by a fitted method. Names are matched against the (flattened) names of the truth.
params	Optional character vector restricting which parameter names to compare. Default NULL uses the intersection of names available in both the truth and the estimates.
tolerance	Positive numeric. Threshold a parameter must fall within to count as recovered. Interpreted as a relative tolerance when <code>relative = TRUE</code> , otherwise an absolute tolerance. Default 0.1.
relative	Logical. If TRUE (default), <code>within_tol</code> compares <code>rel_error <= tolerance</code> ; if FALSE, compares <code>abs_error <= tolerance</code> .

x A recovery_result object.
 ... Further arguments (ignored).
 object A recovery_result object.

Value

An S3 object of class `c("recovery_result", "data.frame")` with one row per compared parameter and columns:

parameter Parameter name (flattened, e.g. `coefs.x1`).
 true True value from the simulation.
 estimate Estimated value.
 abs_error `abs(estimate - true)`.
 rel_error `abs_error / abs(true)`, NA when `true == 0`.
 within_tol Logical recovery flag (see `relative`).

Use `summary.recovery_result` for a one-row scorecard.

Examples

```
# t-test: recover the true Cohen's d
sim <- simulate_ttest(n_a = 200, n_b = 200, mean_a = 0, mean_b = 0.5,
                     sd_a = 1, sd_b = 1, seed = 1)
fit <- t.test(score ~ group, data = sim$data)
d_hat <- unname(fit$estimate[2] - fit$estimate[1]) # mean_b - mean_a
validate_recovery(sim, estimates = c(cohens_d = d_hat))

# Regression: recover known coefficients from lm()
rsim <- simulate_regression(
  coefs = c("(Intercept)" = 1, x1 = 2, x2 = -1.5),
  predictor_sds = c(x1 = 1, x2 = 1), error_sd = 1, n = 2000, seed = 7
)
fit_lm <- lm(y ~ x1 + x2, data = rsim$data)
est <- stats::setNames(coef(fit_lm),
                      paste0("coefs.", names(coef(fit_lm))))
validate_recovery(rsim, estimates = est)
```

validate_sim_params *Validate and Standardize Simulation Parameters*

Description

Validate simulation parameters and set defaults for missing values. Ensures parameter relationships are valid (e.g., `min_na <= max_na`).

Usage

```
validate_sim_params(params)
```

Arguments

params List of simulation parameters. May include:

- seq_length** Maximum sequence length (default: 30).
- n_sequences** Number of sequences to generate (default: 100).
- min_na** Minimum number of NA values per sequence (default: 0).
- max_na** Maximum number of NA values per sequence (default: 5).

Details

The function performs the following validations:

- Sets default values for missing parameters.
- Ensures `max_na <= seq_length`.
- Ensures `min_na <= max_na`.
- Ensures `min_na < seq_length`.

Both new (`seq_length`, `n_sequences`) and old (`max_seq_length`, `num_rows`) parameter names are supported for backward compatibility.

Value

A list of validated parameters with defaults applied.

Examples

```
# Validate with defaults
params <- validate_sim_params(list())

# Validate custom parameters (new names)
params <- validate_sim_params(list(
  seq_length = 50,
  n_sequences = 200,
  min_na = 2,
  max_na = 10
))

# Old parameter names still work
params <- validate_sim_params(list(
  max_seq_length = 50,
  num_rows = 200
))
```

Index

- * **datasets**
 - GLOBAL_NAMES, 26
 - GROUP_REGULATION_ACTIONS, 27
 - learning_states, 30
- analyze_grid_results
 - (summarize_grid_results), 116
- batch_apply, 3, 5
- batch_fit_models, 4, 5
- batch_fit_models(), 119
- calculate_edge_recovery
 - (compare_edge_recovery), 7
- compare_centralities, 6
- compare_centralities(), 14
- compare_edge_recovery, 7
- compare_edge_recovery(), 14
- compare_estimation, 9
- compare_network_estimation, 11
- compare_networks, 13
- compare_networks(), 7, 8
- compare_reliability, 14
- compare_tna_models
 - (compare_network_estimation), 11
- create_param_grid
 - (generate_param_grid), 20
- cross_validate_tna, 17
- evaluate_bootstrap
 - (run_bootstrap_iteration), 39
- export_simulation, 18
- fit_network_model, 5, 19, 65, 107, 112, 114
- generate_group_tna_networks
 - (simulate_group_tna_networks), 63
- generate_param_grid, 20
- generate_probabilities, 22
- generate_probabilities(), 15
- generate_sequence_data
 - (simulate_tna_datasets), 105
- generate_tna_datasets
 - (simulate_tna_datasets), 105
- generate_tna_matrix
 - (simulate_tna_matrix), 108
- generate_tna_networks
 - (simulate_tna_networks), 112
- get_global_names, 23, 27, 61
- get_learning_states, 24, 79, 98, 107, 114
- get_scenario, 25
- GLOBAL_NAMES, 24, 26, 61
- GROUP_REGULATION_ACTIONS, 27
- inject_missingness, 28
- LEARNING_STATES (learning_states), 30
- learning_states, 30
- list_learning_categories, 31, 107
- list_name_regions, 24, 27, 32, 72, 87
- list_scenarios, 25, 32, 52
- list_simulators, 33, 55
- match.arg, 29, 75
- plot.network_estimation, 33
- plot.tna_reliability_comparison, 34
- plot_network_estimation, 35
- plot_sampling_distribution, 36
- plot_tna_comparison, 37
- print.network_estimation, 38
- print.recovery_result
 - (validate_recovery), 122
- print.tna_reliability_comparison, 39
- run_bootstrap_iteration, 39
- run_bootstrap_iteration(), 8
- run_bootstrap_simulation, 42
- run_grid_simulation, 45
- run_network_simulation, 48

run_network_simulation(), [121](#)
 run_sampling_analysis, [50](#)
 run_scenario, [52](#), [121](#)

 sample_tna, [52](#)
 saqr_sim, [18](#), [29](#), [33](#), [52](#), [55–58](#), [67](#), [68](#), [75](#),
 [82](#), [86](#), [93](#), [104](#), [115](#), [121](#), [122](#)
 select_states, [53](#), [107](#)
 seqHMM::simulate_initial_probs(), [22](#)
 seqHMM::simulate_transition_probs(),
 [22](#)
 simulate, [33](#), [55](#)
 simulate_anova, [56](#)
 simulate_clusters, [57](#)
 simulate_correlation, [58](#)
 simulate_data, [33](#), [55](#), [59](#), [81](#)
 simulate_edge_list, [60](#), [88](#)
 simulate_fa, [62](#)
 simulate_group_tna_networks, [63](#), [112](#),
 [114](#)
 simulate_growth, [66](#)
 simulate_hmm, [67](#)
 simulate_htna, [69](#), [84](#), [108](#), [110](#)
 simulate_igraph, [71](#), [88](#)
 simulate_igraph(), [60](#)
 simulate_irt, [74](#)
 simulate_lca, [76](#)
 simulate_long_data, [65](#), [77](#), [89](#), [91](#), [98](#)
 simulate_longitudinal, [80](#)
 simulate_lpa, [82](#)
 simulate_matrix, [70](#), [73](#), [83](#), [88](#), [110](#)
 simulate_mlm, [85](#)
 simulate_mlna(simulate_htna), [69](#)
 simulate_mtna(simulate_htna), [69](#)
 simulate_network, [73](#), [86](#)
 simulate_onehot_data, [79](#), [89](#)
 simulate_prediction, [92](#)
 simulate_regression, [92](#), [93](#)
 simulate_seq_clusters, [94](#)
 simulate_sequences, [79](#), [91](#), [95](#), [107](#), [112](#)
 simulate_sequences(), [60](#)
 simulate_sequences_advanced, [98](#), [99](#)
 simulate_survival, [103](#)
 simulate_tna_datasets, [105](#), [112–114](#)
 simulate_tna_matrix, [70](#), [108](#)
 simulate_tna_network, [73](#), [111](#)
 simulate_tna_networks, [65](#), [79](#), [98](#), [105](#),
 [107](#), [110](#), [112](#), [112](#)
 simulate_ttest, [55](#), [114](#)

 smart_select_states(select_states), [53](#)
 summarize_grid_results, [116](#)
 summarize_networks, [118](#)
 summarize_networks(), [121](#)
 summarize_simulation, [120](#)
 summarize_simulation(), [119](#)
 summary.recovery_result, [123](#)
 summary.recovery_result
 (validate_recovery), [122](#)

 tidy_simulation_results, [18](#), [121](#)

 validate_recovery, [122](#)
 validate_sim_params, [123](#)