Package: clockSim (via r-universe)

February 22, 2025

Title Simulation of the Circadian Clock Gene Network

Version 0.0.0.9002

Description A preconfigured simulation workflow for the circadian clock gene network.

License GPL (>= 3)

Encoding UTF-8

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.2

Imports bench, dde, dplyr, ggplot2, lomb, matrixStats, odin, rlang, tidyr

Suggests knitr, rmarkdown

VignetteBuilder knitr

Config/pak/sysreqs make libicu-dev libssl-dev

Repository https://yeyuan98.r-universe.dev

RemoteUrl https://github.com/yeyuan98/clockSim

RemoteRef HEAD

RemoteSha 3cc8b11701d9f6a0b1a9f6f6257f16ac026225cd

Contents

.compute_validatePair *Trajectory similarity: validate consistency of result matrix pair*

Description

Verify that the two matrices share the same time values and the same states.

Usage

```
.compute_validatePair(res_mat1, res_mat2)
```

Arguments

res_mat1	Trajectory matrix 1, first column must be time while others are states.
res_mat2	Trajectory matrix 2, first column must be time while others are states.

Examples

```
# Internal only; error if check fails.
```

clockSim *clockSim: Simulation of the Circadian Clock Gene Network.*

Description

This package provides preconfigured circadian clock gene network simulation models based on the Odin simulation engine.

Details

Circadian clock is a paradigm for studying negative-positive feedback mechanisms in Biology. Molecular clock, a cell-autonomous transcriptional-translational feedback, is one of the best known gene network oscillators.

Despite widespread interest on molecular clock research, numerical simulation of the clock remains elusive to many if not most wet-lab biologists, which hindered generation of novel hypotheses and efficient exploration of the clock parameter space.

This package aims to provide a low-friction workflow for molecular clock simulation, thereby unleashing the power of in silico exploration to more.

Models:

The mypackage functions ...

`compute_cosine` *Trajectory similarity: Cosine Similarity*

Description

Cosine Similarity is used to characterize how *state direction* of two trajectory $\vec{X}(t), \vec{Y}(t)$ agree with each other. It computes cosine of the angle between the two states at each time.

Usage

```
compute_cosine(res_mat1, res_mat2)
```

Arguments

`res_mat1`	Trajectory matrix 1, first column must be time while others are states.
`res_mat2`	Trajectory matrix 2, first column must be time while others are states.

Details

Formula: $\text{CS}(t^j) = \frac{\Sigma_i X_i(t^j) Y_i(t^j)}{\|X_i(t^j)\| \|Y_i(t^j)\|}$

Value

Cosine similarity of the two trajectory, vector with length = number of time steps.

Examples

```
# Perfect alignment (cos=1)
mat1 <- cbind(time = 1:3, state1 = 1:3, state2 = 4:6)
mat2 <- cbind(time = 1:3, state1 = 2:4, state2 = 5:7)
compute_cosine(mat1, mat2)  # c(1, 1, 1)

# Orthogonal vectors (cos=0)
mat3 <- cbind(time = 1:2, state1 = c(1, 0), state2 = c(0, 1))
mat4 <- cbind(time = 1:2, state1 = c(0, 1), state2 = c(1, 0))
compute_cosine(mat3, mat4)  # c(0, 0)

# Opposite direction (cos=-1)
mat5 <- cbind(time = 1:3, state1 = 1:3)
mat6 <- cbind(time = 1:3, state1 = -1:-3)
compute_cosine(mat5, mat6)  # c(-1, -1, -1)
```

compute_normalize *Trajectory similarity: variable normalization*

Description

Performs per-state normalization by dividing state values by: range (max-min) or sd (standard deviation). none means no normalization is performed.

Usage

```
compute_normalize(res_mat, method = c("none", "range", "sd"), ref_stats = NULL)
```

Arguments

res_mat	Trajectory matrix, first column must be time while others are states.
method	State normalization method, one of "none", "range", "sd".
ref_stats	Optional list containing pre-computed normalization statistics (e.g., from a previous call to compute_normalize). Must include: - For method = "range": mins and maxs vectors - For method = "sd": means and sds vectors

Value

Normalized trajectory matrix with attribute "ref_stats" containing normalization parameters.

Examples

```
mat <- cbind(time = 1:10, matrix(runif(30, 0, 10), ncol = 3))

# Range normalization with automatic stat computation
norm_mat1 <- compute_normalize(mat, "range")

# Reuse stats for another matrix
mat2 <- cbind(time = 1:10, matrix(runif(30, 5, 15), ncol = 3))
norm_mat2 <- compute_normalize(mat2, "range", attr(norm_mat1, "ref_stats"))
```

compute_period *Compute periodicity of time series*

Description

Supports the following methods: fft and lomb. lomb uses the lomb package for computation. Please note that lomb can be resource intensive and significantly slower than fft.

Usage

```
compute_period(ts_vector, ts_time = NULL, method = "fft")
```

Arguments

ts_vector	Numeric vector of time series
ts_time	Numeric vector of time points (if NULL use "step" unit)
method	Period calculation method.

Details

If ts_time is provided, it is passed to the Lomb-Scargle algorithm for unevenly sampled data computation. In this case, lomb method returns period in unit of ts_time. ts_time has no effect on the fft method as it requires even time spacing.

If ts_time is NULL, assume even time spacing and period unit will be in the (implicitly provided) time spacing of the time series vector.

Power, SNR, and p-value of the period detection are method-specific:

1. fft: power = Spectrum power of the peak. \ SNR = (power of peak)/(median power). p-value is not available (NA).
2. lomb: power = LS power of the peak. \ SNR is not available (NA). p-value = LS p-value.

Value

Named vector of length 4 (period, power, snr, p.value).

Examples

```
# Generate a period = 50 sine wave data with some noise (even spacing)
n <- 1000
time <- seq(1, n, by = 1)
ts_data <- sin(2 * pi * time / 50) + rnorm(n, sd = 0.5)
compute_period(ts_data)
compute_period(ts_data, method = "lomb")
# Uneven sampling of the previous data and run lomb method again
s <- sample(1:n, n/3)
compute_period(ts_data[s], time[s], method = "lomb")
```

---

compute_rmse *Trajectory similarity: Root Mean Squared Error RMSE*

Description

RMSE is used to characterize how *absolute* values of two trajectory $\vec{X}(t), \vec{Y}(t)$ agree with each other. There are normalized and default versions:

Usage

```
compute_rmse(res_mat1, res_mat2, normalize = "none")
```

Arguments

res_mat1	Trajectory matrix 1, first column must be time while others are states.
res_mat2	Trajectory matrix 2, first column must be time while others are states.
normalize	Normalization method, one of "none", "range", "sd".

Details

Default (value scales with variable i):

$\text{RMSE}_i = \sqrt{\frac{\Sigma(X_i(t^j) - Y_i(t^j))^2}{N}}$

Normalized (value normalized by range for each variable i):

$\text{NRMSE}_i = \frac{\text{RMSE}}{\text{spread}}$ (spread is customizable)

For normalization, spread is always computed ONLY from res_mat1. res_mat2 is normalized using spread computed from mat1 to still retain the property that RMSE allows comparing *absolute* values of two trajectories.

In this case, normalization is used to prevent numerically large states from dominating RMSE results, giving a more thorough comparison of trajectories.

Value

RMSE of the two trajectory, vector with length = number of input states.

Examples

```
# Perfect agreement (zero error)
mat1 <- cbind(time = 1:3, state1 = c(1, 2, 3), state2 = c(4, 5, 6))
mat2 <- mat1
compute_rmse(mat1, mat2)

# Simple error case
#   state1 = sqrt(mean(c(0,0,0.5)^2)) = 0.29
#   state2 = sqrt(mean(c(0,0.2,0)^2)) = 0.12
mat3 <- cbind(time = 1:3, state1 = c(1, 2, 3.5), state2 = c(4, 5.2, 6))
compute_rmse(mat1, mat3)

# Normalized example (NRMSE = RMSE / spread)
#   state1 = state2 = 1/20 = 0.05
mat4 <- cbind(time = 1:3, state1 = c(10, 20, 30), state2 = c(40, 50, 60))
mat5 <- cbind(time = 1:3, state1 = c(11, 21, 31), state2 = c(41, 51, 61))
compute_rmse(mat4, mat5, "range")  # c(1/20, 1/20) = c(0.05, 0.05)
```

getOdinGen *Get all clock model Odin generator objects*

Description

Refer to package documentation on preconfigured models in this package. This function is intended for advanced usage only; normally calling other helper functions will suffice for simulation.

Usage

```
getOdinGen()
```

Value

Named list of Odin generator R6 objects.

Examples

```
# TODO
```

grid_scan *Running model on a grid of parameters/initial states*

Description

This function is useful for running a large scan of parameter combinations for the same model. Typical use cases are probing stability of an attractor, effect of certain parameters on the system, etc.

Usage

```
grid_scan(
  model_gen,
  grid,
  apply.fn = identity,
  n.core = 2,
  custom.export = NULL,
  ...
)
```

Arguments

model_gen	Odin model generator, see getOdinGen().
grid	Data frame of the parameter grid.
apply.fn	Function to apply before return (e.g., some summary).
n.core	Number of cores for parallel computing.
custom.export	Names of additional variables used by apply.fn.
...	Model $run(...) parameters.

Details

Grid is a data frame whose columns are model parameters. See model$contents() for tunable parameters.

Value

List of model run results.

Examples

```
# TODO
```

---

plot_phase *2-D phase potrait plot*

Description

Provides convenience function to plot simulation trajectory of result from one run in 2-D phase space.

Usage

```
plot_phase(df_result, x, y, time = NULL)
```

Arguments

df_result	Data frame containing results to plot
x	Column to plot as X-axis
y	Column to plot as Y-axis
time	Column to color the trajectory

Value

ggplot object of phase portrait

Examples

```
# TODO
```

`plot_timeSeries` *Time series plot (multifaceted)*

Description

Provides convenience function to plot simulation trajectory of result from one run as time series. Supports plotting multiple states in faceted plot.

Usage

```
plot_timeSeries(df_result, start_time, end_time, sample_time, tick_time, ...)
```

Arguments

`df_result`	Data frame containing results to plot. Must have column $time.
`start_time`	Plot start time.
`end_time`	Plot end time.
`sample_time`	Time interval to subset data.
`tick_time`	X-axis label break interval time (default 2*sample_time).
`...`	... <tidy-select> Columns to plot.

Details

Support the following features:

1. flexible start-end time of the series by `start_time` and `end_time`.
2. row subset of data by a fixed `sample_time` (useful when time step is small for numerical precision).
3. flexible time tick label spacing by `tick_time`.
4. states to plot are selected by `tidy-select`.

Please note that this function does not attempt to do exhaustive sanity check of parameters Make sure yourself that parameters make sense (e.g., end-start time must be longer than sample_time and tick_time, etc.)

Value

ggplot object of faceted time series

Examples

```
# TODO
```

`run_eta` *Run time estimate for an Odin model run.*

Description

Perform test runs of an Odin model with the specified run parameters and return measured run time for planning large-scale simulation repeats and/or model parameter scans.

Usage

```
run_eta(odin_model, ...)
```

Arguments

`odin_model`	An Odin model `R6 <odin_model>`
`...`	Passed to `$run(...)`

Value

Estimated run time and resource measured by test run.

Examples

```
# TODO
```

Index