Package: TreEvo (via r-universe)

August 23, 2024

Type Package

Title Using ABC to Understand Trait Evolution

Version 0.22.1

Date 2023-07-27

Maintainer Brian C. O'Meara <omeara.brian@gmail.com>

Description Various functions for estimating parameters of trait evolution in comparative analyses using Approximate Bayesian Computation.

License GPL

URL https://github.com/bomeara/treevo

BugReports https://github.com/bomeara/TreEvo/issues

Depends R (>= 3.5.0), ape

Imports coda, fastmatch, foreach, graphics, grDevices, MASS, methods, parallel, partitions, phylolm, phytools, pls, RcppZiggurat, rgl, stats, utils, corpcor, mvtnorm, covr

Suggests knitr, paleotree, rmarkdown, testthat, doMC, doParallel

Remotes rdpeng/gpclib

VignetteBuilder knitr

Date/Publication 2012-07-02 16:00:00

Encoding UTF-8

LazyData yes

RoxygenNote 7.2.3

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

RemoteUrl https://github.com/bomeara/treevo

RemoteRef HEAD

RemoteSha f0e13d5bb486b3da0e63e858cd9a00a502fe233d

Contents

`TreEvo-package` *TreEvo–abc for comparative methods*

Description

A package for applying Approximate Bayesian Computation to estimating parameters of trait evolution in comparative analyses.

Details

Package:	TreEvo
Type:	Package
Version:	0.3.3
Date:	2012-07-02
License:	GPL

Author(s)

Brian O'Meara, Barb L. Banbury, David W. Bapst

Maintainer: David Bapst <dwbapst@gmail.com>

Examples

```
# example analysis, using data simulated with TreEvo



set.seed(1)
# let's simulate some data, and then try to infer the parameters using ABC
# get a 20-taxon coalescent tree
tree <- rcoal(20)
# get realistic edge lengths
tree$edge.length <- tree$edge.length*20

genRateExample <- c(0.01)
ancStateExample <- c(10)

#Simple Brownian motion
simCharExample <- doSimulation(
    phy = tree,
    intrinsicFn = brownianIntrinsic,
    extrinsicFn = nullExtrinsic,
    startingValues = ancStateExample, #root state
    intrinsicValues = genRateExample,
    extrinsicValues = c(0),
    generation.time = 100000
    )

# NOTE: the example analyses below sample too few particles,
    # over too few steps, with too few starting simulations
    # - all for the sake of examples that reasonably test the functions

# Please set these values to more realistic levels for your analyses!

resultsBMExample <- doRun_prc(
  phy = tree,
  traits = simCharExample,
  intrinsicFn = brownianIntrinsic,
  extrinsicFn = nullExtrinsic,
  startingPriorsFns = "normal",
  startingPriorsValues = list(c(mean(simCharExample[, 1]), sd(simCharExample[, 1]))),
  intrinsicPriorsFns = c("exponential"),
  intrinsicPriorsValues = list(10),
  extrinsicPriorsFns = c("fixed"),
  extrinsicPriorsValues = list(0),
  generation.time = 100000,
  nRuns = 2,
  nStepsPRC = 3,
  numParticles = 20,
     nInitialSimsPerParam = 10,
  jobName = "examplerun_prc",
  stopRule = FALSE,
  multicore = FALSE,
  coreLimit = 1
  )
```

boxcoxTransformation *Box-Cox Transformation for Simulation Summary Values*

Description

`boxcoxTransformation` Box-Cox transforms summary values from a single simulation, while `boxcoxTransformationMatrix` Box-Cox transforms summary values from multiple simulations. The output of `boxcoxTransformationMatrix` needs to be provided as input for `boxcoxTransformation`.

Usage

```
boxcoxTransformation(summaryValuesVector, boxcoxAddition, boxcoxLambda)

boxcoxTransformationMatrix(summaryValuesMatrix)
```

Arguments

`summaryValuesVector`	Vector of summary statistics from a single simulation
`boxcoxAddition`	Vector of boxcox additions from `boxcoxTransformationMatrix`
`boxcoxLambda`	Vector of boxcox lambda values from `boxcoxTransformationMatrix`
`summaryValuesMatrix`	Matrix of summary statistics from simulations

Value

`boxcoxTransformation` returns a vector of Box-Cox transformed summary statistics from a single observation. `boxcoxTransformationMatrix` returns a matrix of Box-Cox transformed summary statistics with the same dimensions as `summaryValuesMatrix`.

Author(s)

Brian O'Meara and Barb Banbury

Examples

```
set.seed(1)
data(simRunExample)

# example simulation

simDataParallel <- parallelSimulateWithPriors(
  nrepSim = 5,
  multicore = FALSE,
  coreLimit = 1,
  phy = simPhyExample,
  intrinsicFn = brownianIntrinsic,
  extrinsicFn = nullExtrinsic,
  startingPriorsFns = "normal",
  startingPriorsValues = list(c(
     mean(simCharExample[, 1]), sd(simCharExample[, 1]))),
  intrinsicPriorsFns = c("exponential"),
  intrinsicPriorsValues = list(10),
  extrinsicPriorsFns = c("fixed"),
  extrinsicPriorsValues = list(0),
  generation.time = 100000,
  checkpointFile = NULL,
  checkpointFreq = 24,
  verbose = FALSE,
  freevector = NULL, taxonDF = NULL)

nParFree <- sum(attr(simDataParallel, "freevector"))

# separate the simulation results:
  # 'true' generating parameter values from the summary values
summaryValuesMat <- simDataParallel[, -1:-nParFree]

boxTranMat <- boxcoxTransformationMatrix(
  summaryValuesMatrix = summaryValuesMat
  )
boxTranMat

boxcoxTransformation(
  summaryValuesVector = summaryValuesMat[, 1],
  boxcoxAddition = boxTranMat$boxcoxAddition,
  boxcoxLambda = boxTranMat$boxcoxLambda
  )
```

`compareListIPD` *Plot Interparticle Distance between Runs*

Description

This function plots interparticle distance (IPD) between runs for each free parameter.

Usage

```
compareListIPD(particleDataFrameList, verbose = FALSE)
```

Arguments

`particleDataFrameList`	A list composed of multiple `particleDataFrame` objects, as output by `doRun` functions.
`verbose`	Commented screen output.

Value

Produces a plot with IPD between runs per generation.

Author(s)

Brian O'Meara and Barb Banbury

Examples

```
data(simRunExample)

pdfList <- list(resultsBMExample[[1]]$particleDataFrame,
     resultsBoundExample[[1]]$particleDataFrame)

compareListIPD(particleDataFrameList = pdfList, verbose = TRUE)
```

`doRun` *Run Approximate Bayesian Computation for Phylogenetic Comparative Methods*

Description

The `doRun` functions are the main interface for `TreEvo` users to do Approximate Bayesian Computation (ABC) analysis, effectively wrapping the `simulateWithPriors` functions to perform simulations, which themselves are wrappers for the `doSimulation` function. The two current `doRun` functions are `doRun_prc` which applies a partial rejection control (PRC) ABC analysis over multiple generations of simulations, and `doRun_rej` which performs a full rejection ('rej') ABC analysis.

Usage

```
doRun_prc(
  phy,
  traits,
  intrinsicFn,
  extrinsicFn,
  startingPriorsValues,
  startingPriorsFns,
  intrinsicPriorsValues,
  intrinsicPriorsFns,
  extrinsicPriorsValues,
  extrinsicPriorsFns,
  numParticles = 300,
  nStepsPRC = 5,
  nRuns = 2,
  nInitialSims = NULL,
  nInitialSimsPerParam = 100,
  generation.time = 1000,
  TreeYears = max(branching.times(phy)) * 1e+06,
  multicore = FALSE,
  coreLimit = NA,
  multicoreSuppress = FALSE,
  standardDevFactor = 0.2,
  epsilonProportion = 0.7,
  epsilonMultiplier = 0.7,
  validation = "CV",
  scale = TRUE,
  variance.cutoff = 95,
  stopRule = FALSE,
  stopValue = 0.05,
  maxAttempts = Inf,
  diagnosticPRCmode = FALSE,
  jobName = NA,
  saveData = FALSE,
  verboseParticles = TRUE
)

doRun_rej(
  phy,
  traits,
  intrinsicFn,
  extrinsicFn,
  startingPriorsValues,
  startingPriorsFns,
  intrinsicPriorsValues,
  intrinsicPriorsFns,
  extrinsicPriorsValues,
  extrinsicPriorsFns,
  generation.time = 1000,
  TreeYears = max(branching.times(phy)) * 1e+06,
  multicore = FALSE,
  coreLimit = NA,
  validation = "CV",
  scale = TRUE,
  nInitialSims = NULL,
  nInitialSimsPerParam = 100,
  variance.cutoff = 95,
  standardDevFactor = 0.2,
  jobName = NA,
  abcTolerance = 0.1,
  checkpointFile = NULL,
  checkpointFreq = 24,
  savesims = FALSE
)
```

Arguments

`phy`	A phylogenetic tree, in package `ape`'s `phylo` format.
`traits`	Data matrix with rownames identical to `phy@tip.label`. If a vector, `traits` will be coerced to a matrix, with element names as rownames.
`intrinsicFn`	Name of (previously-defined) function that governs how traits evolve within a lineage, regardless of the trait values of other taxa.
`extrinsicFn`	Name of (previously-defined) function that governs how traits evolve within a lineage, based on their own ('internal') trait vlaue and the trait values of other taxa.
`startingPriorsValues`	A list of the same length as the number of prior distributions specified in `startingPriorsFns` (for starting values, this should be one prior function specified for each trait - thus one for most univariate trait analyses), with each element of the list a vector the same length as the appropriate number of parameters for that prior distribution (1 for `"fixed"`, 2 for `"uniform"`, 2 for `"normal"`, 2 for `"lognormal"`, 2 for `"gamma"`, 1 for `"exponential"`).
`startingPriorsFns`	Vector containing names of prior distributions to use for root states: can be one of `"fixed"`, `"uniform"`, `"normal"`, `"lognormal"`, `"gamma"`, `"exponential"`.
`intrinsicPriorsValues`	A list of the same length as the number of prior distributions specified in `intrinsicPriorsFns` (one prior function specified for each parameter in the intrinsic model), with each element of the list a vector the same length# as the appropriate number of parameters for that prior distribution (1 for `"fixed"`, 2 for `"uniform"`, 2 for `"normal"`, 2 for `"lognormal"`, 2 for `"gamma"`, 1 for `"exponential"`).
`intrinsicPriorsFns`	Vector containing names of prior distributions to use for intrinsic function parameters: can be one of `"fixed"`, `"uniform"`, `"normal"`, `"lognormal"`, `"gamma"`, `"exponential"`.
`extrinsicPriorsValues`	A list of the same length as the number of prior distributions specified in `extrinsicPriorsFns` (one prior function specified for each parameter in the extrinsic model), with each element of the list a vector the same length as the appropriate number of parameters for that prior distribution (1 for `"fixed"`, 2 for `"uniform"`, 2 for `"normal"`, 2 for `"lognormal"`, 2 for `"gamma"`, 1 for `"exponential"`).

`extrinsicPriorsFns`	Vector containing names of prior distributions to use for extrinsic function parameters: can be one of `"fixed"`, `"uniform"`, `"normal"`, `"lognormal"`, `"gamma"`, `"exponential"`.
`numParticles`	Number of accepted particles per PRC generation.
`nStepsPRC`	Number of PRC generations to run.
`nRuns`	Number of independent PRC runs to be performed, each consisting of independent sets of initial simulations and PRC generations. Note that runs are run *sequentially*, and not in parallel, as the generation of particles within each run makes use of the multicore functionality. If `nRuns` is greater than 1, the output from `doRun_prc` will be a list object composed of multiple output lists, as described.
`nInitialSims`	Number of initial simulations used to calibrate particle rejection control algorithm. If not given, this will be a function of the number of freely varying parameters.
`nInitialSimsPerParam`	If `nInitialSims` is not given by the user, the number of initial simulations performed to calibrate the particle rejection algorithm will instead be the number of free parameters in the model multiplied by `nInitialSimsPerParam`.
`generation.time`	The number of years per generation. This sets the coarseness of the simulation; if it's set to 1000, for example, the population's trait values change every 1000 calendar years. Note that this is in calendar years (see description for argument `TreeYears`), and not in millions of years (as is typical for dated trees in macroevolutionary studies). Thus, if a branch is 1 million-year time-unit long, and a user applies the default `generation.time = 1000`, then 1000 evolutionary changes will be simulated along that branch. See documentation for `doSimulation` for further details.
`TreeYears`	The amount of calendar time from the root to the furthest tip. Most trees in macroevolutionary studies are dated with branch lengths in units of millions of years, and thus the default for this argument is `max(branching.times(phy)) * 1e6`. If your tree has the most recent tip at time zero (i.e., the modern day), this would be the same as the root age of the tree. If your branch lengths are not in millions of years, you should alter this argument. Otherwise, leave this argument alone. See documentation for `doSimulation` for further details.
`multicore`	Whether to use multicore, default is `FALSE`. If `TRUE`, one of two suggested packages must be installed, either `doMC` (for UNIX systems) or `doParallel` (for Windows), which are used to activate multithreading. If neither package is installed, this function will fail if `multicore = TRUE`.
`coreLimit`	Maximum number of cores to be used.
`multicoreSuppress`	This argument suppresses the multicore code and will apply a plain vanilla serial `for()` loop instead of `doPar`. Mainly to be used for developer diagnostic purposes.
`standardDevFactor`	Standard deviation for mutating states each time a new particle is generated in a PRC generation.

`epsilonProportion`	Sets tolerance for initial simulations.
`epsilonMultiplier`	Sets tolerance on subsequent PRC generations.
`validation`	Character argument controlling what validation procedure is used by `plsr`. Default is `"CV"` for cross-validation.
`scale`	This argument is passed to `plsr`. It may be a numeric vector, or logical. If numeric vector, the input is scaled by dividing each variable with the corresponding element of scale. If `scale = TRUE`, the inpus is scaled by dividing each variable by its sample standard deviation. If cross-validation is selected (the default for `returnPLSModel`), scaling by the standard deviation is done for every segment.
`variance.cutoff`	Minimum threshold percentage of variance explained for the number of components included in the final PLS model fit. This value is a percentage and must be between 0 and 100. Default is 95 percent.
`stopRule`	If `TRUE`, an analysis will be terminated prior to set number of generations if the ratio of all parameters from one generation to the next falls below the `stopValue`.
`stopValue`	Threshold value for terminating an analysis prior to `nStepesPRC`.
`maxAttempts`	Maximum number attempts made in `while` loop within the PRC algorithm. If reached, the algorithm will terminate with an error message. By default, this is infinite, and thus there is no effective limit without user intervention.
`diagnosticPRCmode`	If `TRUE` (*not* the default), the function will be very noisy about characteristics of the PRC algorithm as it runs.
`jobName`	Optional job name.
`saveData`	Option to save various run information during the analysis, including summary statistics from analyses, output to external .Rdata and .txt files.
`verboseParticles`	If `TRUE` (the default), a large amount of information about parameter estimates and acceptance of particles is output to console via `message` as `doRun_prc` runs.
`abcTolerance`	Proportion of accepted simulations.
`checkpointFile`	Optional file name for checkpointing simulations.
`checkpointFreq`	Saving frequency for checkpointing.
`savesims`	Option to save individual simulations, output to a .Rdata file.

Details

Both `doRun` functions take an input phylogeny (`phy`), observed trait data set (`traits`), models (`intrinsicFn`, `extrinsicFn`), and priors (`startingPriorsValues`, `startingPriorsFns`, `intrinsicPriorsValues`, `intrinsicPriorsFns`, `extrinsicPriorsValues`, `extrinsicPriorsFns`). Pulling from the priors, it simulates an initial set of simulations (`nInitialSims`). This set of simulations is boxcox transformed , and a PLS regression (see `methodsPLS`) is performed for each free parameter to determine the most informative summary statistics (using `variance.cutoff`). The euclidean distance is calculated between each each initial simulation's most informative summary statistics and the input observed data.

Following that step, the approach of the two functions diverge drastically.

For function `doRun_rej`, those simulations whose most informative summary statistics fall below `abcTolerance` are kept as accepted 'particles' (simulations runs), describing the posterior distribution of parameters. No additional generations of simulations are performed.

Coversely, function `doRun_prc` performs an ABC-PRC analysis, which follows a much more complicated algorithm with additional generations. In PRC, tolerance is set based on `epsilonProportion` and `epsilonMultiplier`, and the analysis begins with generation 1 and continues through a number of generations equal to `nStepsPRC`. Single simulation runs ('particles') are accepted if the distance of the most informative summary stats to the original data is less than this tolerance. A generation is complete when enough particles (`numParticles`) have been accepted. These particles make up the distribution for the next generation. The accepted particles from the final generation describe the posterior distributions of parameters.

Value

The output of these two functions are lists, composed of multiple objects, which differ slightly in their content among the two functions. For `doRun_prc`, the output is:

input.data Input variables: `jobName`, `nTaxa` (number of taxa), `nInitialSims`, `generation.time`, `TreeYears`, `timeStep`, `totalGenerations`, `epsilonProportion`, `epsilonMultiplier`, `nRuns` (number of runs), `nStepsPRC`, `numParticles`, `standardDevFactor`.

priorList List of prior distributions. This is used for doing post-analysis comparisons between prior and posterior distributions, such as with function `plotPosteriors`.

particleDataFrame DataFrame with information from each simulation, including generation, attempt, `id`, `parentid`, distance, weight, and parameter states.

toleranceVector Final tolerance vector.

phy Input phylogeny.

traits Input traits.

simTime Processor time for initial simulations.

time.per.gen Processor time for subsequent generations.

postSummary Summarizes the posterior distribution from the final generation for all free parameters, giving the mean, standard deviation and Highest Posterior Density (at a 0.8 alpha) for each parameter.

If `nRuns` is greater than 1, the output from `doRun_prc` will be a list object composed of multiple output lists, as described.

For `doRun_rej`, the output is:

input.data Input variables: `jobName`, `nTaxa` (number of taxa), `nInitialSims`, `generation.time`, `TreeYears`, `timeStep`, `totalGenerations`, `epsilonProportion`, `epsilonMultiplier`, `nRuns` (number of runs), `nStepsPRC`, `numParticles`, `standardDevFactor`.

priorList List of prior distributions. This is used for doing post-analysis comparisons between prior and posterior distributions, such as with function `plotPosteriors`.

phy Input phylogeny.

traits Input traits.

trueFreeValuesANDSummaryValues Parameter estimates and summary stats from all sims.

simTime Processor time for simulations.

abcDistancesRaw Euclidean distances for each simulation and free parameter.

particleDataFrame DataFrame with information from each simulation, including generation, attempt, id, parentid, distance, weight, and parameter states.

postSummary Summarizes the posterior distribution from the final generation for all free parameters, giving the mean, standard deviation and Highest Posterior Density (at a 0.8 alpha) for each parameter.

parMeansList A list of parameter means for both fixed and unfixed parameters, sorted into a list of three vectors (starting, intrinsic, extrinsic).

Author(s)

Brian O'Meara and Barb Banbury

References

Sisson et al. 2007, Wegmann et al. 2009

Examples

```
set.seed(1)
data(simRunExample)

# NOTE: the example analyses below sample too few particles,
    # over too few steps, with too few starting simulations
    # - all for the sake of examples that demonstrate these
    # within a reasonable time-frame

## Please set these values to more realistic levels for your analyses!

resultsPRC <- doRun_prc(
  phy = simPhyExample,
  traits = simCharExample,
  intrinsicFn = brownianIntrinsic,
  extrinsicFn = nullExtrinsic,
  startingPriorsFns = "normal",
  startingPriorsValues = list(c(mean(simCharExample[, 1]),
         sd(simCharExample[, 1]))),
  intrinsicPriorsFns = c("exponential"),
  intrinsicPriorsValues = list(10),
  extrinsicPriorsFns = c("fixed"),
  extrinsicPriorsValues = list(0),
  generation.time = 10000,
  nRuns = 2,
  nStepsPRC = 3,
  numParticles = 20,
  nInitialSimsPerParam = 10,
  jobName = "examplerun_prc",
  stopRule = FALSE,
  multicore = FALSE,
  coreLimit = 1
  )

resultsPRC

#one should make sure priors are uniform with doRun_rej!

resultsRej <- doRun_rej(
    phy = simPhyExample,
    traits = simCharExample,
    intrinsicFn = brownianIntrinsic,
    extrinsicFn = nullExtrinsic,
    startingPriorsFns = "normal",
    startingPriorsValues = list(c(mean(simCharExample[, 1]),
         sd(simCharExample[, 1]))),
    intrinsicPriorsFns = c("exponential"),
    intrinsicPriorsValues = list(10),
    extrinsicPriorsFns = c("fixed"),
    extrinsicPriorsValues = list(0),
    generation.time = 10000,
    jobName = "examplerun_rej",
    abcTolerance = 0.05,
    multicore = FALSE,
    coreLimit = 1
    )

resultsRej
```

doSimulation *Discrete-Time Character Simulation*

Description

The function doSimulation evolves continuous characters under a discrete time process. These functions are mainly used as internal components, generating simulations within ABC analyses using the doRun functions. See *Note* below.

Usage

```
doSimulation(
  phy = NULL,
  intrinsicFn,
  extrinsicFn,
  startingValues,
  intrinsicValues,
  extrinsicValues,
  generation.time = 1000,
  TreeYears = max(branching.times(phy)) * 1e+06,
  timeStep = NULL,
  returnAll = FALSE,
  taxonDF = NULL,
  checkTimeStep = TRUE
)
```

Arguments

`phy`	A phylogenetic tree, in package `ape`'s `phylo` format.
`intrinsicFn`	Name of (previously-defined) function that governs how traits evolve within a lineage, regardless of the trait values of other taxa.
`extrinsicFn`	Name of (previously-defined) function that governs how traits evolve within a lineage, based on their own ('internal') trait vlaue and the trait values of other taxa.
`startingValues`	State at the root.
`intrinsicValues`	Vector of values corresponding to the parameters of the intrinsic model.
`extrinsicValues`	Vector of values corresponding to the parameters of the extrinsic model.
`generation.time`	The number of years per generation. This sets the coarseness of the simulation; if it's set to 1000, for example, the population's trait values change every 1000 calendar years. Note that this is in calendar years (see description for argument `TreeYears`), and not in millions of years (as is typical for dated trees in macroevolutionary studies). Thus, if a branch is 1 million-year time-unit long, and a user applies the default `generation.time = 1000`, then 1000 evolutionary changes will be simulated along that branch. See documentation for `doSimulation` for further details.
`TreeYears`	The amount of calendar time from the root to the furthest tip. Most trees in macroevolutionary studies are dated with branch lengths in units of millions of years, and thus the default for this argument is `max(branching.times(phy)) * 1e6`. If your tree has the most recent tip at time zero (i.e., the modern day), this would be the same as the root age of the tree. If your branch lengths are not in millions of years, you should alter this argument. Otherwise, leave this argument alone. See documentation for `doSimulation` for further details.
`timeStep`	This value corresponds to the length of intervals between discrete evolutionary events ('generations') simulated along branches, relative to a rescaled tree where the root to furthest tip distance is 1. For example, `timeStep = 0.01` of would mean 100 (i.e., 1 / 0.01) evolutionary changes would be expected to occur from the root to the furthest tip. (Note that the real number simulated will be much less, because simulations start over at each branching node.) Ideally, `timeStep` (or its effective value, via other arguments) should be as short as is computationally possible. Typically `NULL` by default and determined internally as follows: `timeStep = generation.time / TreeYears`. Can be provided a value as an alternative to using arguments `generation.time` and `TreeYears`, which would then be overridden. See documentation for `doSimulation` for further details.

`returnAll`	If `TRUE`, the output returned is a `data.frame` containing the values at each node from the simulation.
`taxonDF`	A data.frame containing data on nodes (both tips and internal nodes) output by various internal functions. Can be supplied as input to spead up repeated calculations, but by default is `NULL`, which instead forces a calculation from input `phy`.
`checkTimeStep`	If `TRUE`, warnings will be issued if `TimeStep` is too short.

Details

The phylogenetic tree used is rescaled such that the distance from the root to the furthest tip is rescaled to equal 1 time-unit, and it is this rescaled edge lengths to with arguments `timeStep` refers to. Typically, this will be determined though as a ratio of `TreeYears` (which is the number of calendar years constituing the root-to-furthest-tip distance, and is determined by default as if the user had supplied a tree with edge lengths in time-units of 1 time-unit = 1 million years), and `generation.time`, which gives the length of `timeSteps` in calendar years (e.g. `generation.time = 1000` means an evolutionary change in trait values every 1000 years). Note that the real number of trait change events simulated may be less less, because simulations start over at each branching node, but intervals between changes should be so fine that this should be negligible (related, the results should be independent of your choice for `generation.time` or `timeStep`). We recommend that the effective `timeStep` should be as short as is computationally possible. `SaveRealParams` is useful for tracking the *real* true values if simulating data to test the performance of ABC analyses. It is not useful for ABC analyses of empirical data.

Value

If `returnAll = FALSE` (the default), this function returns a data frame of species character (tip) values in the tree, as a single column named 'states' with rownames reflecting the taxon names given in `phy$tip.label`. If `returnAll = TRUE`, the raw `data.frame` from the simulation will instead be returned.

Note

The `simulateWithPriors` functions are effectively the engine that powers the `doRun` functions, while the `doSimulation` functions are the pistons within the `simulateWithPriors` engine. In general, most users will just drive the car - they will just use `doRun`, but some users may want to use `simulateWithPriors` or `doSimulation` functions to do various simulations.

Author(s)

Brian O'Meara and Barb Banbury

Examples

```
set.seed(1)
tree <- rcoal(20)
# get realistic edge lengths
```

```
tree$edge.length <- tree$edge.length*20

#Simple Brownian motion

char <- doSimulation(
    phy = tree,
    generation.time = 100000,
    intrinsicFn = brownianIntrinsic,
    extrinsicFn = nullExtrinsic,
    startingValues = c(10), #root state
    intrinsicValues = c(0.01),
    extrinsicValues = c(0))

#Character displacement model with minimum bound

char <- doSimulation(
    phy = tree,
    generation.time = 100000,
    intrinsicFn = boundaryMinIntrinsic,
    extrinsicFn = ExponentiallyDecayingPushExtrinsic,
    startingValues = c(10), #root state
    intrinsicValues = c(0.05, 10, 0.01),
    extrinsicValues = c(0, .1, .25))
```

`extrinsicModels` *Extrinsic Character Evolution Models*

Description

Functions describing various models of 'extrinsic' evolution (i.e. evolutionary processes dependent on factors extrinsic to the evolving lineage, such as environmental change, or other evolving lineages that interact with the lineage in question (competitors, predators, etc).

Usage

```
nullExtrinsic(params, selfstates, otherstates, timefrompresent)

nearestNeighborDisplacementExtrinsic(
  params,
  selfstates,
  otherstates,
  timefrompresent
)

everyoneDisplacementExtrinsic(params, selfstates, otherstates, timefrompresent)

ExponentiallyDecayingPushExtrinsic(
  params,
  selfstates,
  otherstates,
  timefrompresent
)
```

Arguments

`params`	A vector containing input parameters for the given model (see *Description* below on what parameters).
`selfstates`	Vector of current trait values for the population of interest. May be multiple for some models, but generally expected to be only a single value. Multivariate `TreEvo` is not yet supported.
`otherstates`	Matrix of current trait values for all concurrent taxa/populations other than the one of interest, with one row for each taxon, and a column for each trait. May be multiple states per taxa/populations for some models, but generally expected to be only a single value. Multivariate `TreEvo` is not yet supported.
`timefrompresent`	The amount of time from the present - generally ignored except for time-dependent models.

Details

The following extrinsic models are:

`nullExtrinsic` describes a model of no extrinsic character change. It has no parameters, really.

`nearestNeighborDisplacementExtrinsic` describes a model of extrinsic trait evolution where character values of a focal taxon depend on the values of closest relatives on the tree (e.g. competitive exclusion). The input parameters for this model are: `nearestNeighborDisplacementExtrinsic` with parameters `params = sigma, springK, maximumForce`

`everyoneDisplacementExtrinsic` describes a model of extrinsic trait evolution where the character values of a focal taxon depend on the values of all co-extant relatives on the simulated tree. The input parameters for this model are: `everyoneDisplacementExtrinsic` with parameters `params = sigma, springK, maximumForce`

`ExponentiallyDecayingPushExtrinsic` describes a model of extrinsic trait evolution where the character values of a focal taxon is 'pushed' away from other taxa with similar values, but the force of that 'push' exponentially decays as lineages diverge and their character values become less similar. The input parameters for this model are: `ExponentiallyDecayingPushExtrinsic` with parameters `params = sigma, maximumForce, halfDistance`

Value

A vector of values representing character displacement of that lineage over a single time step.

Author(s)

Brian O'Meara and Barb Banbury

See Also

Intrinsic models are described at `intrinsicModels`.

Examples

```
set.seed(1)
# Examples of simulations with various extrinsic models (and null intrinsic model)
tree <- rcoal(20)
# get realistic edge lengths
tree$edge.length <- tree$edge.length*20

#No trait evolution except due to
       # character displacement due to nearest neighbor taxon
char <- doSimulation(
    phy = tree,
    intrinsicFn = nullIntrinsic,
    extrinsicFn = nearestNeighborDisplacementExtrinsic,
    startingValues = c(10), #root state
    intrinsicValues = c(0),
    extrinsicValues = c(0.1, 0.1, 0.1),
 generation.time = 100000)

#Similarly, no trait evolution except due to
       # character displacement from all other taxa in the clade
char <- doSimulation(
    phy = tree,
    intrinsicFn = nullIntrinsic,
    extrinsicFn = everyoneDisplacementExtrinsic,
    startingValues = c(10), #root state
    intrinsicValues = c(0),
    extrinsicValues = c(0.1, 0.1, 0.1),
    generation.time = 100000)

# A variant where force of character displacement decays exponentially
        # as lineages become more different
char <- doSimulation(
    phy = tree,
    intrinsicFn = nullIntrinsic,
    extrinsicFn = ExponentiallyDecayingPushExtrinsic,
    startingValues = c(10), #root state
    intrinsicValues = c(0),
    extrinsicValues = c(0.1, 0.1, 2),
    generation.time = 100000)
```

`getBMRatePrior` *Get Brownian Motion Rate Prior*

Description

This function automatically calculates prior distributions for the rate of trait evolution under the Brownian Motion (BM) model on a discrete time-scale, at a given `timeStep`, in the sense that that variable is used with other `TreEvo` functions like `doRun_prc`.

Usage

```
getBMRatePrior(phy, traits, timeStep, verbose = TRUE)
```

Arguments

`phy`	A phylogenetic tree, in package `ape`'s `phylo` format.
`traits`	Data matrix with rownames identical to `phy@tip.label`. If a vector, `traits` will be coerced to a matrix, with element names as rownames.
`timeStep`	time in a single iteration of the discrete-time simulation
`verbose`	If `TRUE`, gives messages about how the simulation is progessing via `message`.

Details

Returns a matrix of prior values that can be used in the `doRun` functions. Builds on functions in `phylolm` to estimate distribution.

Value

Returns a matrix of prior values

Author(s)

Brian O'Meara and Barb Banbury

Examples

```
data(simRunExample)

#timeStep = 0.1 -> effectively ~100 steps over the tree length

getBMRatePrior(phy = simPhyExample, traits = simCharExample,
   timeStep = 0.01, verbose = TRUE)
```

`highestDensityInterval` *Highest Density Interval*

Description

This function calculates highest density intervals (HDIs) for a given univariate vector. Typically, this is done to obtain highest posterior density (HPD) for each freely varying parameter estimated in the posterior of a Bayesian MCMC analysis. If these intervals are calculated for more than one variable, they are referred to instead as regions.

Usage

```
highestDensityInterval(
  dataVector,
  alpha,
  coda = FALSE,
  verboseMultimodal = TRUE,
  stopIfFlat = TRUE,
  ...
)
```

Arguments

`dataVector`	A vector of data, which can be reasonably assumed to represent independent, identically-distributed random variables, such as estimates of a parameter from a Bayesian MCMC.
`alpha`	The threshold used for defining the highest density frequency cut-off. If the highest density interval is applied to a Bayesian MCMC posterior sample, then the interval is effectively calculated for this value as a posterior probability density.
`coda`	Default is `FALSE`. If `TRUE`, unimodal highest density regions will instead be calculated using `HPDinterval` from package `coda`, which is similar to the function `quantile` in that it calculates only a single interval.
`verboseMultimodal`	If `TRUE`, the function will print a message indicating when the inferred highest density interval is discontinuous, and thus likely reflects that the supplied data is multimodal.
`stopIfFlat`	If `TRUE` (the default), the function will terminate and return an error if the distribution appears to be flat, with a high number of tied values in the posterior. If `FALSE`, the function will instead attempt to move forward, while returning a warning instead. The latter option is mainly implemented so internal calls to `highestDensityInterval` do not block otherwise long-running analyses.
`...`	Additional arguments passed to `density`. A user may want to mess with this to adjust bandwidth, et cetera.

Details

By default, HDI calculation is preformed by fitting a kernal density estimate (KDE) via R function `density` with default bandwidth, rescaling the kernal to a density, sorting intervals along the KDE by that density, and then summing these values from largest to smallest, until the desired alpha is reached. This algorithm is quick, and accounts for potentially multimodal distributions, or those with complex shapes, unlike unimodal intervals, such as quantiles, or the `HPDinterval` in package `coda`.

Alternatively, a user can opt to use function `HPDinterval` from package `coda` to calculate highest density intervals. These will work as long as the data has a single mode - data with multiple modes may have overly large quantiles (to encompass those multiple modes), resulting in overly wide estimates of coverage.

References

Hyndman, R. J. 1996. Computing and Graphing Highest Density Regions. *The American Statistician* 50(2):120-126.

See Also

`HPDinterval` in package `coda` for an alternative used by older versions of `TreEvo` which cannot properly handle multimodal distributions.

Examples

```
set.seed(444)

# let's imagine we have some variable with
    # an extreme bimodal distribution
z <- sample(c(rnorm(50, 1, 2), rnorm(100, 50, 3)))
hist(z)

# now let's say we want to know the what sort of values
# are reasonably consistent with this distribution

# for example, let's say we wanted the ranges within
# which 80% of our data occurs

# one way to do this would be a quantile
    # two tailed 80% quantiles
quantile(z, probs = c(0.1, 0.9))

# that seems overly broad - there's essentially no density
# in the central valley - but we want to exclude values found there!
# A value of 30 doesn't match this data sample, right??

# the problem is that all quantile methods are essentially based on
# the empirical cumulative distribution function - which is monotonic
# (as any cumulutative function should be), and thus
# quantiles can only be a single interval
# A different approach is to use density from stats
density(z)
# we could then take the density estimates for
# particular intervals, rank-order them, and
# then cumulative sample until we reach
# our desired probability density (alpha)

# let's try that
alpha <- 0.8
zDensOut <- density(z)
zDensity <- zDensOut$y/sum(zDensOut$y)
inHPD <- cumsum(-sort(-zDensity)) <= alpha
# now reorder
inHPD <- inHPD[order(order(-zDensity))]
colDens <- rep(1, length(zDensity))
colDens[inHPD] <- 2
# and let's plot it, with colors
plot(zDensOut$x, zDensity, col = colDens)

# and we can do all that (except the plotting)
    # with highestDensityInterval
highestDensityInterval(z, alpha = 0.8)

#############################
# example with output from doRun_prc
data(simRunExample)

# we'll use summarizePosterior, which just automates picking
  # the last generation, and freely-varying parameters for HDRs

# alpha = 0.95
summarizePosterior(
   resultsBMExample[[1]]$particleDataFrame,
   alpha = 0.95)

# you might be tempted to use alphas like 95%, but with bayesian statistics
# we often don't sample the distribution well enough to know
# its shape to exceeding detail. alpha = 0.8 may be more reasonable.
summarizePosterior(
   resultsBMExample[[1]]$particleDataFrame,
   alpha = 0.8)
```

`intrinsicModels` *Intrinsic Character Evolution Models*

Description

Functions describing various models of ’intrinsic’ evolution (i.e. evolutionary processes intrinsic to the evolving lineage, independent of other evolving lineages (competitors, predators, etc).

Usage

```
nullIntrinsic(params, states, timefrompresent)

brownianIntrinsic(params, states, timefrompresent)

boundaryIntrinsic(params, states, timefrompresent)

boundaryMinIntrinsic(params, states, timefrompresent)

boundaryMaxIntrinsic(params, states, timefrompresent)

autoregressiveIntrinsic(params, states, timefrompresent)

maxBoundaryAutoregressiveIntrinsic(params, states, timefrompresent)

minBoundaryAutoregressiveIntrinsic(params, states, timefrompresent)

autoregressiveIntrinsicTimeSlices(params, states, timefrompresent)

autoregressiveIntrinsicTimeSlicesConstantMean(params, states, timefrompresent)

autoregressiveIntrinsicTimeSlicesConstantSigma(params, states, timefrompresent)
```

Arguments

`params`	A vector containing input parameters for the given model (see *Description* below on what parameters).
`states`	Vector of current trait values for a taxon. May be multiple for some models, but generally expected to be only a single value. Multivariate `TreEvo` is not yet supported.
`timefrompresent`	The amount of time from the present - generally ignored except for time-dependent models.

Details

The following intrinsic models are:

`nullIntrinsic` describes a model of no intrinsic character change. It has no parameters, really.

`brownianIntrinsic` describes a model of intrinsic character evolution via Brownian motion. The input parameters for this model are: `boundaryIntrinsic` with parameters `params = sigma`

`boundaryIntrinsic` describes a model of intrinsic character evolution where character change is restricted above a minimum and below a maximum threshold. The input parameters for this model are: `boundaryMinIntrinsic` with parameters `params = sigma, minimum, maximum`

`boundaryMinIntrinsic` describes a model of intrinsic character evolution where character change is restricted above a minimum threshold. The input parameters for this model are: `boundaryMinIntrinsic` with parameters `params = sigma, minimum` `autoregressiveIntrinsic` describes a model of intrinsic character evolution. New character values are generated after one time step via a discrete-time OU process. The input parameters for this model are: `autoregressiveIntrinsic` with `params` = `sigma (sigma)`, `attractor (character mean)`, `attraction (alpha)`

`minBoundaryAutoregressiveIntrinsic` describes a model of intrinsic character evolution. New character values are generated after one time step via a discrete-time OU process with a minimum bound. The input parameters for this model are: `MinBoundaryAutoregressiveIntrinsic` with parameters `params` = `sigma (sigma)`, `attractor (character mean)`, `attraction (alpha)`, `minimum`

`autoregressiveIntrinsicTimeSlices` describes a model of intrinsic character evolution. New character values are generated after one time step via a discrete-time OU process with differing means, sigma, and attraction over time. In the various *TimeSlices* models, time threshold units are in time before present (i.e., 65 could be 65 MYA). The last time threshold should be 0. The input parameters for this model are: `autoregressiveIntrinsicTimeSlices` with parameters `params` = `sd-1 (sigma-1)`,`attractor-1 (character mean-1)`, `attraction-1 (alpha-1)`, `time threshold-1`,`sd-2 (sigma-2)`, `attractor-2 (character mean-2)`, `attraction-2 (alpha-2)`, `time threshold-2`

`autoregressiveIntrinsicTimeSlicesConstantMean` describes a model of intrinsic character evolution. New character values are generated after one time step via a discrete-time OU process with differing sigma and attraction over time The input parameters for this model are: `autoregressiveIntrinsicTimeSlice` with parameters `params` = `sd-1 (sigma-1)`, `attraction-1 (alpha-1)`, `time threshold-1`, `sd-2 (sigma-2)`,`attraction-2 (alpha-2)`, `time threshold-2`, `attractor (character mean)`

`autoregressiveIntrinsicTimeSlicesConstantSigma` describes a model of intrinsic character evolution. New character values are generated after one time step via a discrete-time OU process with differing means and attraction over time. The input parameters for this model are: `autoregressiveIntrinsicTimeSlicesConstantSigma` with parameters `params` = `sigma (sigma)`,`attractor-1 (character mean-1)`, `attraction-1 (alpha-1)`, `time threshold-1`,`attractor-2 (character mean-2)`, `attraction-2 (alpha-2)`, `time threshold-2`

Value

A vector of values representing character displacement of that lineage over a single time step.

Author(s)

Brian O'Meara and Barb Banbury

See Also

Another intrinsic model with multiple optima is described at `multiOptimaIntrinsic`. Extrinsic models are described at `abcmodels.extrinsic`.

Examples

```
set.seed(1)
# Examples of simulations with various intrinsic models (and null extrinsic model)
tree <- rcoal(20)
# get realistic edge lengths
tree$edge.length <- tree$edge.length*20
```

```
#Simple Brownian motion Intrinsic Model
char <- doSimulation(
    phy = tree,
    intrinsicFn = brownianIntrinsic,
    extrinsicFn = nullExtrinsic,
    startingValues = c(10), #root state
    intrinsicValues = c(0.01),
    extrinsicValues = c(0),
    generation.time = 100000)

# Simple model with BM, but a minimum bound at 0, max bound at 15
char <- doSimulation(
    phy = tree,
    intrinsicFn = boundaryIntrinsic,
    extrinsicFn = nullExtrinsic,
    startingValues = c(10), #root state
    intrinsicValues = c(0.01, 0, 15),
    extrinsicValues = c(0),
    generation.time = 100000)

# Autoregressive (Ornstein-Uhlenbeck) model
       # with minimum bound at 0
char <- doSimulation(
    phy = tree,
    intrinsicFn = minBoundaryAutoregressiveIntrinsic,
    extrinsicFn = nullExtrinsic,
    startingValues = c(10), #root state
    intrinsicValues = c(0.01, 3, 0.1, 0),
    extrinsicValues = c(0),
    generation.time = 100000)

# Autoregressive (Ornstein-Uhlenbeck) model
       # with max bound at 1
char <- doSimulation(
    phy = tree,
    intrinsicFn = maxBoundaryAutoregressiveIntrinsic,
    extrinsicFn = nullExtrinsic,
    startingValues = c(10), #root state
    intrinsicValues = c(0.01, 3, 0.1, 1),
    extrinsicValues = c(0),
    generation.time = 100000)
```

landscapeFPK_Intrinsic

Intrinsic Trait Evolution Model for a Macroevolutionary Landscape on a Fokker-Planck-Kolmogorov Potential Surface (Boucher et al., 2018)

Description

This function describes a discrete-time Fokker-Planck-Kolmogorov Potential Surface model (the FPK model for short), described for trait macroevolution by Boucher et al. (2018), and included here in `TreEvo` for use as an intrinsic trait model. This model can be used to describe a complex landscape of hills and valleys for a bounded univariate trait space.

Usage

```
landscapeFPK_Intrinsic(params, states, timefrompresent, grainScaleFPK = 100)

getTraitBoundsFPK(traitData)

plot_landscapeFPK_model(
  params,
  grainScaleFPK = 1000,
  traitName = "Trait",
  plotLandscape = TRUE
)
```

Arguments

`params`	A vector containing input parameters for the given model (see *Description* below on what parameters).
`states`	Vector of current trait values for a taxon. May be multiple for some models, but generally expected to be only a single value. Multivariate `TreEvo` is not yet supported.
`timefrompresent`	The amount of time from the present - generally ignored except for time-dependent models.
`grainScaleFPK`	To calculate the potential-surface landscape, the trait space is discretized into fine intervals, and the potential calculated for each interval. The scale of this discretization can be controlled with this argument, which specifies the number of intervals used (default is 1000 intervals).
`traitData`	A set of trait data to calculate distant bounds from for use with this model.
`traitName`	The name given to the trait, mainly for use in the macroevolutionary landscape plot.
`plotLandscape`	If `TRUE`, the estimated macroevolutionary landscape is plotted by the function.

Details

The FPK model is a four parameter model for describing the evolution of a trait without reference to the traits of other taxa (i.e. an intrinsic model in the terminology of package `TreEvo`). Three of these parameters are used to describe the shape of the landscape, which dictates a lineage's overall deterministic evolutionary trajectory on the landscape, while the fourth parameter (sigma) is a dispersion parameter that describes the rate of unpredictable, stochastic change.

The bounds on the trait space are treated as two additional parameters, but these are intended in the model as described by Boucher et al. to be treated as nuisance parameters. Boucher et al.

intentionally fix these bounds at a far distance beyond the range of observed trait values, so in order to ensure they have as little effect on the evolutionary trajectories as possible.

The discrete time Fokker-Planck-Kolmogorov model used here describes a landscape specific to a population at a particular time, with a particular trait value. This landscape represents a potential surface, where height corresponds to a tendency to change in that direction. This allows for multiple optima, and assigns different heights to those optima, which represent the current attraction for a population to move toward that trait value. The shape of this potential surface is the macroevolutionary landscape. which Boucher et al describe the shape of using a fourth-order polynomial with the cubic component removed:

$$V(x) = ax^4 + bx^2 + cx$$

Where x is the trait values within the bounded interval - for simplicity, these are internally rescaled to sit within the arbitrary interval `(-1.5 : 1.5)`. The parameters thus that describe the landscape shape are the coefficients *a*, *b*, and *c*. To calculate the landscape, the trait space is discretized into fine intervals, and the potential calculated for each interval. The scale of this discretization can be controlled by the user.

Note that if the landscape has a single peak, or if the potential surface is flat, the model effectively collapses to Brownian Motion, or Ornstein-Uhlenbeck with a single optima. To (roughly) quote Boucher et al: ‘Finally, note that both BM and the OU model are special cases of the FPK model: BM corresponds to V(x)=0 and OU to V(x)=((alpha/sigma^2)*x^2)-((2*alpha*theta/(sigma^2))*x).’

Value

A vector of values representing character displacement of that lineage over a single time step.

Author(s)

David W. Bapst, loosely based on studying the code for function `Sim_FPK` from package `BBMV`.

References

Boucher, F. C., V. Demery, E. Conti, L. J. Harmon, and J. Uyeda. 2018. A General Model for Estimating Macroevolutionary Landscapes. *Systematic Biology* 67(2):304-319.

See Also

An alternative approach in `TreEvo` to estimating a macroevolutionary landscape with multiple optima is the intrinsic model `multiOptimaIntrinsic`, however this model requires an *a priori* choice on the number of optima and the assumption that the optima have similar attractor strength. Other intrinsic models are described at `intrinsicModels`.

Examples

```
set.seed(444)
traitData<-rnorm(100,0,1)
# need traits to calculate bounds
bounds<-getTraitBoundsFPK(traitData)
```

```
# pick a value at random
trait<-0

# two peak symmetric landscape example
params<-c(
 a=2,
 b=-4,
 c=0,
 sigma=1,
 bounds)

plot_landscapeFPK_model(params)

# simulate under this model - simulated trait DIVERGENCE
landscapeFPK_Intrinsic(params=params, states=trait, timefrompresent=NULL)

# simulate n time-steps, repeat many times, plot results
repeatSimSteps<-function(params,trait,nSteps){
 for(i in 1:nSteps){
 # add to original trait value to get new trait value
  trait<-trait+landscapeFPK_Intrinsic(
   params=params, states=trait, timefrompresent=NULL)
   }
 trait
 }
repSim<-replicate(30,repeatSimSteps(params,trait,20))
hist(repSim,main="Simulated Trait Values")


# uneven two peak symmetric landscape example
params<-c(
 a=2,
 b=-4,
 c=0.3,
 sigma=1,
 bounds)
plot_landscapeFPK_model(params)

# simulate under this model - simulated trait DIVERGENCE
landscapeFPK_Intrinsic(params=params, states=trait, timefrompresent=NULL)

repSim<-replicate(30,repeatSimSteps(params,trait,20))
hist(repSim,main="Simulated Trait Values")
```

---

methodsPLS *Fitting Univariate Partial Least Squares Models to Free Parameters in ABC*

Description

Function `returnPLSModel` fits a PLS regression (using `plsr`) individually to each freely varying parameter of a model, unlike a true multivariate PLS regression. A secondary step than limits the number of components to those that explain some minimum cumulative percentage of variance (see argument `variance.cutoff`). For ABC, this seems to result in much better results, without one parameter dominating the combined variance.

Usage

```
returnPLSModel(
  trueFreeValuesMatrix,
  summaryValuesMatrix,
  validation = "CV",
  scale = TRUE,
  variance.cutoff = 95,
  verbose = TRUE,
  segments = min(10, nrow(summaryValuesMatrix) - 1),
  ...
)

PLSTransform(summaryValuesMatrix, pls.model)
```

Arguments

`trueFreeValuesMatrix`	Matrix of true free values from simulations.
`summaryValuesMatrix`	Matrix of summary statistics from simulations.
`validation`	Character argument controlling what validation procedure is used by `plsr`. Default is `"CV"` for cross-validation.
`scale`	This argument is passed to `plsr`. It may be a numeric vector, or logical. If numeric vector, the input is scaled by dividing each variable with the corresponding element of scale. If `scale = TRUE`, the inpus is scaled by dividing each variable by its sample standard deviation. If cross-validation is selected (the default for `returnPLSModel`), scaling by the standard deviation is done for every segment.
`variance.cutoff`	Minimum threshold percentage of variance explained for the number of components included in the final PLS model fit. This value is a percentage and must be between 0 and 100. Default is 95 percent.
`verbose`	If `TRUE`, helpful warning messages will be made when you make questionable decisions.
`segments`	Number of segments of data used for crossvalidaton by `pls::cvsegments`. The number of segments cannot exceed the number of simulations. The default number of segments as set for normal use of `plsr` is 10, which leads to issues when a trial analysis uses fewer than 10 simulations. Instead, we will pass an alternative value for the number of segments - either 10, or the number of rows in `summaryValuesMatrix`. Thus, this is a default value of `min(10,`

	`nrow(summaryValuesMatrix))`, or can be changed by the user. A hard minimum of 3 is required.
`...`	Additional arguments, passed to `plsr`.
`pls.model`	Output from `returnPLSModel`.

Details

Function `PLSTransform` uses results from a Partial Least Squares (PLS) model fit with `returnPLSModel` to transform summary values.

Value

Function `returnPLSModel` returns a PLS model, and function `PLSTransform` returns transformed summary statistics.

Author(s)

Brian O'Meara and Barb Banbury

See Also

Function `returnPLSModel` effectively wraps function `plsr` from package `pls` (see documentation at `mvr`).

Examples

```
set.seed(1)
simPhyExample <- rcoal(20)
simPhyExample$edge.length <- simPhyExample$edge.length * 20

# example simulation

nSimulations <- 6

simDataParallel <- parallelSimulateWithPriors(
   nrepSim = nSimulations,
   multicore = FALSE,
   coreLimit = 1,
   phy = simPhyExample,
   intrinsicFn = brownianIntrinsic,
   extrinsicFn = nullExtrinsic,
   startingPriorsFns = "normal",
   startingPriorsValues = list(
       c(mean(simCharExample[, 1]),
       sd(simCharExample[, 1]))),
   intrinsicPriorsFns = c("exponential"),
   intrinsicPriorsValues = list(10),
   extrinsicPriorsFns = c("fixed"),
   extrinsicPriorsValues = list(0),
   generation.time = 10000,
   checkpointFile = NULL,
    checkpointFreq = 24,
    verbose = FALSE,
    freevector = NULL,
    taxonDF = NULL
    )

nParFree <- sum(attr(simDataParallel, "freevector"))

# separate the simulation results:
    # 'true' generating parameter values from the summary values
trueFreeValuesMat <- simDataParallel[, 1:nParFree]
summaryValuesMat <- simDataParallel[, -1:-nParFree]

PLSmodel <- returnPLSModel(
    trueFreeValuesMatrix = trueFreeValuesMat,
    summaryValuesMatrix = summaryValuesMat,
    validation = "CV",
    scale = TRUE,
    variance.cutoff = 95 ,
    segments = nSimulations
    )

PLSmodel

PLSTransform(
    summaryValuesMatrix = summaryValuesMat,
    pls.model = PLSmodel
    )
```

`multiOptimaIntrinsic` *Single Evolutionary Regime Model with Multiple Optima*

Description

This model describes an evolutionary process where multiple optima exist. Lineages are attracted to an optima in their vicinity (this is handled as a stochastic process where which attract affects which population at a given point in time is weighted with respect to the proximity of a given population's trait value to the given optima), but as a lineage approaches the attractor that controls it, it may experience less directional evolution in the direction of the optima, and overall show more variation. This framework allows for the model to describe the evolution with respect to multiple optima simultaneously, without needing to treat individual transitions between optima (or macroevolutionary 'regimes') as a separate parameter. Thus, the optima exist within a single evolutionary regime, with only a scaling parameter present that allows more or less frequent transitions between optima by altering the impact of proximity to optima.

Usage

```
multiOptimaIntrinsic(params, states, timefrompresent)
```

Arguments

`params`	A vector containing input parameters for the given model (see *Description* below on what parameters).
`states`	Vector of current trait values for a taxon. May be multiple for some models, but generally expected to be only a single value. Multivariate `TreEvo` is not yet supported.
`timefrompresent`	The amount of time from the present - generally ignored except for time-dependent models.

Details

`multiOptimaIntrinsic` describes a model of intrinsic character evolution with multiple regimes. New character values are generated after one time step via a discrete-time OU process with a particular optima assigned to a particular regime, and each time-step a lineage has some finite probability of switching to a new regime, and being drawn to that regime's optima. The chance of a lineage being drawn to a particular optima is based on proximity to that optima, but the chance of switching to another regime is never completely negligible. The strength of the draw to the optima, the attraction strength, 'alpha', is the same for all regimes (all optima). This model has *n* input parameters: `multiOptimaIntrinsic` with `params = sigma` (rate of dispersion), `alpha` (strength of attraction to an optima), `rho` (an exponent scaling the weighting of distance to optima – this parameter will control the probability of a lineage switching optima), and, finally, *n* (two or more) `theta` values, which are the optima for the different macroevolutionary adaptive regimes.

Our biological interpretation for this model is a scenario in which optima represent fixed phenotypic trait values, conveying maximum adaptive benefit relative to neighboring values in trait space. The proximity of a population to an optima makes it more likely to fall under the influence of that regime, and thus experience directional selection in the direction of that optima. However, as there are multiple optima, a lineage might be influenced by multiple nearby optima over its evolutionary history, rather than simply the closest trait optimum. Lineages equidistant between multiple optima should be equally likely to be drawn to any specific optima, and populations in general should experience the influence of nearby optima inversely relative to their distance from the optima. Thus, a lineage very close to an optimum would show a large variance in its trait values as it circles the adaptive plateau, with infrequent but sudden, far-spanning movements toward a different optimum.

Value

A vector of values representing character displacement of that lineage over a single time step.

Author(s)

David W. Bapst

See Also

An alternative approach in `TreEvo` to estimating a macroevolutionary landscape with multiple optima is the Fokker-Planck-Kolmogorov (FPK) model, which can be fit with `landscapeFPK_Intrinsic`. This model does not assume a set number of optima nor that they have similar attractor strength, but parameters may be difficult to interpret in isolation, and fitting this model may be slower with

TreEvo due to necessary linear algebra transformations. Other intrinsic models are described at intrinsicModels.

Examples

```
# three optima model, with strong attraction
set.seed(1)
params<-c(
 sigma=0.1,
 alpha=0.7,
 rho=1,
 theta=c(-20,20,50)
 )

multiOptimaIntrinsic(params=params, states=0, timefrompresent=NA)

# simulate n time-steps, repeat many times, plot results
repeatSimSteps<-function(params,trait=0,nSteps){
 for(i in 1:nSteps){
 # add to original trait value to get new trait value
  trait<-trait+multiOptimaIntrinsic(
   params=params, states=trait, timefrompresent=NA)
  }
 trait
 }
repSim<-replicate(300,repeatSimSteps(params,trait=0,100))
hist(repSim,main="Simulated Trait Values",breaks=20)


# same model above, with more switching between optima
set.seed(1)
params<-c(
 sigma=0.1,
 alpha=0.7,
 rho=0.5,
 theta=c(-20,20,50)
 )

multiOptimaIntrinsic(params=params, states=0, timefrompresent=NA)

# simulate n time-steps, repeat many times, plot results
repeatSimSteps<-function(params,trait=0,nSteps){
 for(i in 1:nSteps){
 # add to original trait value to get new trait value
  trait<-trait+multiOptimaIntrinsic(
   params=params, states=trait, timefrompresent=NA)
  }
 trait
 }
repSim<-replicate(300,repeatSimSteps(params,trait=0,100))
hist(repSim,main="Simulated Trait Values",breaks=20)
```

pairwiseESS *Pairwise Effective Sample Size*

Description

This function calculates Effective Sample Size (ESS) on results. The ESS algorithm performs best when results are from multiple runs.

Usage

```
pairwiseESS(inputData)
```

Arguments

`inputData`	The input dataset can be a single data frame or a list of data frames.

Value

Returns a matrix with ESS values of all pairwise runs.

Author(s)

Brian O'Meara and Barb Banbury

Examples

```
data(simRunExample)

# this will give a warning
pairwiseESS(resultsBMExample[[1]]$particleDataFrame)

# ESS should be calculated over multiple runs
pairwiseESS(resultsBMExample)

# you can also manually assemble a list of particleDataFrame tables
    # and use this as the input
inputList <- list(
    resultsBMExample[[2]]$particleDataFrame,
    resultsBMExample[[1]]$particleDataFrame
    )
pairwiseESS(inputList)
```

`pairwiseKS` *Pairwise Kolmogorov-Smirnov test*

Description

This function calculates Kolmogorov-Smirnov on results.

Usage

```
pairwiseKS(particleDataFrameList)
```

Arguments

`particleDataFrameList`	An object of type list, composed of particleDataFrames from separate analyses.

Value

Returns a matrix with Kolmogorov-Smirnov values of all pairwise runs.

Author(s)

Brian O'Meara and Barb Banbury

Examples

```
data(simRunExample)

pdfList <- list(
   Brownian = resultsBMExample[[1]]$particleDataFrame,
   Bounded = resultsBoundExample[[1]]$particleDataFrame
   )

pairwiseKS(particleDataFrameList = pdfList)
```

`parentOffspringPlots` *Plotting Parent-Offspring Particles*

Description

This function uses the `particleDataFrame` output by TreEvo ABC analyses and plots parent-offspring particles from generation to generation.

Usage

```
parentOffspringPlots(particleDataFrame)
```

Arguments

`particleDataFrame`	`particleDataFrame` output by TreEvo ABC analyses.

Details

Each parameter is plotted twice for parent-offspring relationships through the generations. In the top row, particles are plotted as a measure of distance to the observed data; the farther away the particle the bigger the circle. In the bottom row, particles are plotted as a measure of their weights; larger circles are closer to observed data and therefore carry more weight in the analysis.

As of version 0.6.0, rejected particles are not saved for outputting by the parallelized algorithm, and thus they are no longer displayed by this function.

Value

Creates a set of parent-offspring plots.

Author(s)

Brian O'Meara and Barb Banbury

Examples

```
data(simRunExample)
parentOffspringPlots(resultsBMExample[[1]]$particleDataFrame)
```

`plotABC_3D` *3D ABCplots*

Description

Plot posterior density distribution for each generation in 3d plot window

Usage

```
plotABC_3D(
  particleDataFrame,
  parameter,
  show.particles = "none",
  plot.parent = FALSE,
  realParam = FALSE,
  realParamValues = NA
)
```

Arguments

`particleDataFrame`	A `particleDataFrame` object, as found among the output from `doRun` functions.
`parameter`	Column number of parameter of interest from `particleDataFrame`.
`show.particles`	Option to show particles on 3d plot as "none" or as a function of "weights" or "distance".
`plot.parent`	Option to plot lines on the floor of the 3d plot to show particle parantage.
`realParam`	Option to display real parameter value as a solid line, also must give actual value for this (realParamValues). Note: this should only be done with simulated data where real param values are recorded.
`realParamValues`	Value for `realParam`.

Details

This opens a new interactive 3d plotting window and plots the posterior density distribution of accepted particles from each generation. Several options are available to add to the plot: plotting particles by weight or distance, plotting particle parantage, and plotting the real parameter values (if known).

As of version 0.6.0, rejected particles are not saved for outputting by the parallelized algorithm, and thus they are no longer displayed by this function, unlike previous versions.

Note

This function requires access to the function `triangulate` and the `as` method for class `gpc.poly` from package `gpclib`. As of 08-22-19, this package was not available from CRAN as a Windows binary, and thus this function is likely unavailable to many (if not all) Windows users.

This function also requires the package `rgl`, which is usually easier to obtain than package `gpclib`, but may not be buildable on some UNIX workstations.

Author(s)

Barb Banbury

Examples

```
# need to check for required suggested packages
if(requireNamespace("gpclib", quietly = TRUE) & requireNamespace("rgl", quietly = TRUE)){

 data(simRunExample)
 plotABC_3D(
     particleDataFrame = resultsBMExample[[1]]$particleDataFrame,
     parameter = 7,
     show.particles = "none",
     plot.parent = FALSE,
     realParam = FALSE,
     realParamValues = NA
```

```
    )

}
```

plotPosteriors *Plot posterior distributions*

Description

For each free parameter in the posterior, this function creates a plot of the distribution of values estimated in the last generation. This function can also be used to visually compare against true (generating) parameter values in a simulation.

Usage

```
plotPosteriors(
  particleDataFrame,
  priorsList,
  realParam = FALSE,
  realParamValues = NA
)
```

Arguments

particleDataFrame	A particleDataFrame object returned by TreEvo ABC analyses, can be a single data frame or a list of data frames. If the particleDataFrame is a list of separate TreEvo runs, posteriors will be layered over each other to check for repeatability. The relative gray-scale of posterior distributions in the plot depends on their total number of runs.
priorsList	A priorList list object, returned by TreEvo doRun functions, where each element is itself a list of length two, composed of the name for the prior function (e.g. 'uniform', and a vector of the parameters for that prior distribution). Can be a single such list, or a list of priorList lists from a series of analyses (i.e. a list of depth 3). If the priorList is a list of lists (if it is found to be a list of depth 3, only the first list will be evaluated, and all other lists will be ignored. In other words, priors are expected to be identical for all runs considered by this function, such the choice of using the list of priors from the first analysis in the list is acceptable for all comparisons.
realParam	If TRUE, this function will plot line segments where real parameter values are known. (Usually only true when simulated data is analyzed.)
realParamValues	Values for real parameters, include a value for each parameter (including fixed values). Otherwise should be NA.

Value

Returns a plot for each free parameter.

Note

realParam and realParamValues should only be used with simulated data, where the true values are known.

Author(s)

Barb Banbury and Brian O'Meara

See Also

plotPrior for a set of functions for visualizing and summarizing prior and posterior distributions, including a visual comparison for single parameters.

Examples

```
data(simRunExample)

# make a list of particleDataFrames to plot multiple runs
resultsPDFlist <- lapply(resultsBMExample, function(x) x$particleDataFrame)

plotPosteriors(
   resultsPDFlist,
   priorsList = resultsBMExample[[1]]$priorList,
   realParam = TRUE,
   realParamValues = c(ancStateExample, genRateExample)
   )
```

---

plotPriorPost *Plotting, Summarizing and Comparing the Prior and Posterior Distributions*

Description

Assorted functions for visualizing and summarizing the prior and posterior probability distributions associated with ABC analyses.

Usage

```
plotPrior(
  priorFn = match.arg(arg = priorFn, choices = c("fixed", "uniform", "normal",
    "lognormal", "gamma", "exponential"), several.ok = FALSE),
  priorVariables,
  plotQuants = TRUE,
  plotLegend = TRUE
)

plotUnivariatePosteriorVsPrior(
  posteriorCurve,
  priorCurve,
  label = "parameter",
  trueValue = NULL,
  ...
)

getUnivariatePriorCurve(
  priorFn,
  priorVariables,
  nPoints = 1e+05,
  from = NULL,
  to = NULL,
  alpha = 0.8,
  coda = FALSE,
  verboseMultimodal = TRUE,
  ...
)

getUnivariatePosteriorCurve(
  acceptedValues,
  from = NULL,
  to = NULL,
  alpha = 0.8,
  coda = FALSE,
  verboseMultimodal = TRUE,
  ...
)
```

Arguments

`priorFn`	Prior Shape of the distribution; one of either "fixed", "uniform", "normal", "lognormal", "gamma", or "exponential".
`priorVariables`	Variables needed to describe the shape of the distribution, dependent on `priorFn`: **priorFn = "uniform"** priorVariables = c(min, max) **priorFn = "normal"** priorVariables = c(mean, standard deviation) **priorFn = "lognormal"** priorVariables = c(mean, standard deviation) **priorFn = "gamma"** priorVariables = c(shape, scale) **priorFn = "exponential"** priorVariables = c(rate)
`plotQuants`	If `TRUE`, plots line segments at the quantiles
`plotLegend`	If `TRUE`, plots legend box with quantile values
`posteriorCurve`	Kernal density estimates for the posterior distribution from `getUnivariatePosteriorCurve`.
`priorCurve`	Kernal density estimates for the prior distribution from `getUnivariatePriorCurve`.
`label`	Horizontal X-axis label for the plot.
`trueValue`	True parameter value, if any such exists and is known (usually only true of simulations).

`...`	For `getUnivariatePriorCurve` and `getUnivariatePosteriorCurve`, this can contain additional arguments passed to `density`, for use in both calculating the kernal density estimate for finding the curve, and for estimating the highest density interval. A user may want to mess with this to adjust bandwidth, et cetera. For `plotUnivariatePosteriorVsPrior`, this passes additional commands to the initial call `plot`, and thus can set things like a `main` plotting title, among other things.
`nPoints`	Number of points to draw.
`from`	Lower bound, if any. By default this is `NULL` and thus ignored.
`to`	Upper bound, if any. By default this is `NULL` and thus ignored.
`alpha`	The threshold used for defining the highest density frequency cut-off. If the highest density interval is applied to a Bayesian MCMC posterior sample, then the interval is effectively calculated for this value as a posterior probability density.
`coda`	Default is `FALSE`. If `TRUE`, unimodal highest density regions will instead be calculated using `HPDinterval` from package `coda`, which is similar to the function `quantile` in that it calculates only a single interval.
`verboseMultimodal`	If `TRUE`, the function will print a message indicating when the inferred highest density interval is discontinuous, and thus likely reflects that the supplied data is multimodal.
`acceptedValues`	Vector of accepted particle values.

Details

Function `plotPrior` visualizes the shape of various prior probability distributions available in TreEvo ABC analyses, while `getUnivariatePriorCurve` returns density coordinates and summary statistics from user-selected prior probability distribution.

Similarly, function `getUnivariatePosteriorCurve` returns density coordinates and summary statistics from the posterior distribution of an ABC analysis.

Both `getUnivariatePriorCurve` and `getUnivariatePosteriorCurve` also calculate the highest density intervals for their respective parameters, using the function `highestDensityInterval`.

Function `plotUnivariatePosteriorVsPrior` plots the univariate density distributions from the prior and posterior against each other for comparison, along with the highest density intervals (HDI) for both.

The summaries calculated from `getUnivariatePriorCurve` and `getUnivariatePosteriorCurve` are used as the input for `plotUnivariatePosteriorVsPrior`, hence the relationship of these functions to each other, and why they are listed together here.

Value

`plotPrior` and `plotUnivariatePosteriorVsPrior` produce plots of the respective distributions (see above).

`getUnivariatePriorCurve` returns a list of `x` and `y` density coordinates, mean, and the highest density intervals (HDI) for their respective distribution.

`getUnivariatePosteriorCurve` does the same for a posterior sample of parameter estimates, returning a list of x and y density coordinates, mean, and the highest posterior density intervals (HPD).

Author(s)

Brian O'Meara and Barb Banbury

See Also

Highest posterior densities are calculated via `highestDensityInterval`.

`plotPosteriors` Plots multiple posteriors against their priors and potential known values.

Examples

```
data(simRunExample)

# examples with plotPrior

plotPrior(
    priorFn = "exponential",
    priorVariables = c(10))

plotPrior(
    priorFn = "normal",
    priorVariables = c(1, 2))

plotPrior(
    priorFn = "gamma",
    priorVariables = c(2, .2),
    plotQuants = FALSE,
    plotLegend = FALSE)

# examples of getting density coordinates and
  # summary statistics from distributions

priorKernal <- getUnivariatePriorCurve(
    priorFn = "normal",
    priorVariables = c(28, 2),
    nPoints = 100000,
    from = NULL,
    to = NULL,
    alpha = 0.95)

postKernal <- getUnivariatePosteriorCurve(
    acceptedValues =
        resultsBMExample[[1]]$particleDataFrame$starting_1,
    from = NULL,
    to = NULL,
    alpha = 0.95)

priorKernal
postKernal
```

```
# let's compare this (supposed) prior
  # against the posterior in a plot

plotUnivariatePosteriorVsPrior(
    posteriorCurve = postKernal,
    priorCurve = priorKernal,
    label = "parameter",
    trueValue = NULL)

# cool!
```

simRunExample *Example Analysis Output of a Simulated Dataset*

Description

Simulated 30-taxon coalescent tree and simulated character data from a Brownian Motion intrinsic model (`brownianIntrinsic`), with saved generating parameters. Character data was generated under this model using `doSimulation`. Also includes results from an example analysis.

Format

Loading the `simRunExample` example dataset adds seven new objects to the namespace:

`simPhyExample` A simulated 30-tip coalescent phylogeny in typical `phylo` format.

`ancStateExample` The starting ancestral value, used for generating the simulated continuous trait data.

`genRateExample` The true rate of trait change under Brownian Motion, used for generating the simulated continuous trait data.

`simCharExample` The output of `doSimulation` on `simPhyExample`, under the model `brownianIntrinsic`. composed of just the simulated trait values as a one-column matrix with row names indicating tip labels, as desired by `doRun` functions.

`resultsBMExample` The results of `doRun_prc`, under the generating model of `brownianIntrinsic`

`resultsBoundExample` The results of `doRun_prc`, under the incorrect model of `boundaryMinIntrinsic`

The objects `resultsBMExample` and `resultsBoundExample` are lists composed of a number of elements (see the documentation for the `doRun_prc` function for more detail).

Examples

```
data(simRunExample)

# ...things to do with this data?

###################
```

```
# This data was generated using this process:

## Not run:

library(TreEvo)

set.seed(1)
simPhyExample <- rcoal(20)
# get realistic edge lengths
simPhyExample$edge.length <- simPhyExample$edge.length*20

# plot with time axis (root is about ~15 Ma)
plot(simPhyExample)
axisPhylo()

genRateExample <- c(0.001)
ancStateExample <- c(10)

#Simple Brownian motion
simCharExample <- doSimulation(
    phy = simPhyExample,
    intrinsicFn = brownianIntrinsic,
    extrinsicFn = nullExtrinsic,
    startingValues = ancStateExample, #root state
    intrinsicValues = genRateExample,
    extrinsicValues = c(0),
    generation.time = 10000
    )

resultsBMExample <- doRun_prc(
    phy = simPhyExample,
    traits = simCharExample,
    intrinsicFn = brownianIntrinsic,
    extrinsicFn = nullExtrinsic,
    startingPriorsFns = "normal",
    startingPriorsValues = list(c(mean(simCharExample[, 1]), sd(simCharExample[, 1]))),
    intrinsicPriorsFns = c("exponential"),
    intrinsicPriorsValues = list(10),
    extrinsicPriorsFns = c("fixed"),
    extrinsicPriorsValues = list(0),
    generation.time = 10000,
    nRuns = 2,
    nStepsPRC = 3,
    numParticles = 20,
    nInitialSimsPerParam = 10,
    jobName = "examplerun_prc",
    stopRule = FALSE,
    multicore = FALSE,
    verboseParticles = TRUE,
    coreLimit = 1
    )
```

```
resultsBoundExample <- doRun_prc(
    phy = simPhyExample,
    traits = simCharExample,
    intrinsicFn = boundaryMinIntrinsic,
    extrinsicFn = nullExtrinsic,
    startingPriorsFns = "normal",
    startingPriorsValues = list(c(mean(simCharExample[, 1]), sd(simCharExample[, 1]))),
    intrinsicPriorsFns = c("exponential", "normal"),
    intrinsicPriorsValues = list(10,c(-10,1)),
    extrinsicPriorsFns = c("fixed"),
    extrinsicPriorsValues = list(0),
    generation.time = 10000,
    nRuns = 2,
    nStepsPRC = 3,
    numParticles = 20,
    nInitialSimsPerParam = 10,
    jobName = "examplerun_prc_bound",
    stopRule = FALSE,
    multicore = FALSE,
    verboseParticles = TRUE,
    coreLimit = 1
    )

rm(.Random.seed)
save.image(file = "simRunExample.rdata")
if(interactive()){savehistory("simRunExample.Rhistory")}


## End(Not run)
```

---

simulateWithPriors *Simulate data for initial TreEvo analysis*

Description

The simulateWithPriors function pulls parameters from prior distributions and conducts a single simulation of continuous trait evolution (using the doSimulation function), returning useful summary statistics for ABC. parallelSimulateWithPriors is a wrapper function for simulateWithPriors that allows for multithreading and checkpointing. This family of functions is mostly used as internal components, generating simulations within ABC analyses using the doRun functions. See *Note* below.

Usage

```
simulateWithPriors(
  phy = NULL,
  intrinsicFn,
  extrinsicFn,
  startingPriorsFns,
  startingPriorsValues,
  intrinsicPriorsFns,
  intrinsicPriorsValues,
  extrinsicPriorsFns,
  extrinsicPriorsValues,
  generation.time = 1000,
  TreeYears = max(branching.times(phy)) * 1e+06,
  timeStep = NULL,
  giveUpAttempts = 10,
  verbose = FALSE,
  checks = TRUE,
  taxonDF = NULL,
  freevector = NULL
)

parallelSimulateWithPriors(
  nrepSim,
  multicore,
  coreLimit,
  phy,
  intrinsicFn,
  extrinsicFn,
  startingPriorsFns,
  startingPriorsValues,
  intrinsicPriorsFns,
  intrinsicPriorsValues,
  extrinsicPriorsFns,
  extrinsicPriorsValues,
  generation.time = 1000,
  TreeYears = max(branching.times(phy)) * 1e+06,
  timeStep = NULL,
  checkpointFile = NULL,
  checkpointFreq = 24,
  verbose = TRUE,
  checkTimeStep = TRUE,
  verboseNested = FALSE,
  freevector = NULL,
  taxonDF = NULL,
  giveUpAttempts = 10
)
```

Arguments

`phy`	A phylogenetic tree, in package `ape`'s `phylo` format.
`intrinsicFn`	Name of (previously-defined) function that governs how traits evolve within a lineage, regardless of the trait values of other taxa.

`extrinsicFn` Name of (previously-defined) function that governs how traits evolve within a lineage, based on their own ('internal') trait vlaue and the trait values of other taxa.

`startingPriorsFns`
Vector containing names of prior distributions to use for root states: can be one of `"fixed"`, `"uniform"`, `"normal"`, `"lognormal"`, `"gamma"`, `"exponential"`.

`startingPriorsValues`
A list of the same length as the number of prior distributions specified in `startingPriorsFns` (for starting values, this should be one prior function specified for each trait - thus one for most univariate trait analyses), with each element of the list a vector the same length as the appropriate number of parameters for that prior distribution (1 for `"fixed"`, 2 for `"uniform"`, 2 for `"normal"`, 2 for `"lognormal"`, 2 for `"gamma"`, 1 for `"exponential"`).

`intrinsicPriorsFns`
Vector containing names of prior distributions to use for intrinsic function parameters: can be one of `"fixed"`, `"uniform"`, `"normal"`, `"lognormal"`, `"gamma"`, `"exponential"`.

`intrinsicPriorsValues`
A list of the same length as the number of prior distributions specified in `intrinsicPriorsFns` (one prior function specified for each parameter in the intrinsic model), with each element of the list a vector the same length# as the appropriate number of parameters for that prior distribution (1 for `"fixed"`, 2 for `"uniform"`, 2 for `"normal"`, 2 for `"lognormal"`, 2 for `"gamma"`, 1 for `"exponential"`).

`extrinsicPriorsFns`
Vector containing names of prior distributions to use for extrinsic function parameters: can be one of `"fixed"`, `"uniform"`, `"normal"`, `"lognormal"`, `"gamma"`, `"exponential"`.

`extrinsicPriorsValues`
A list of the same length as the number of prior distributions specified in `extrinsicPriorsFns` (one prior function specified for each parameter in the extrinsic model), with each element of the list a vector the same length as the appropriate number of parameters for that prior distribution (1 for `"fixed"`, 2 for `"uniform"`, 2 for `"normal"`, 2 for `"lognormal"`, 2 for `"gamma"`, 1 for `"exponential"`).

`generation.time`
The number of years per generation. This sets the coarseness of the simulation; if it's set to 1000, for example, the population's trait values change every 1000 calendar years. Note that this is in calendar years (see description for argument `TreeYears`), and not in millions of years (as is typical for dated trees in macroevolutionary studies). Thus, if a branch is 1 million-year time-unit long, and a user applies the default `generation.time = 1000`, then 1000 evolutionary changes will be simulated along that branch. See documentation for `doSimulation` for further details.

`TreeYears` The amount of calendar time from the root to the furthest tip. Most trees in macroevolutionary studies are dated with branch lengths in units of millions of years, and thus the default for this argument is `max(branching.times(phy)) * 1e6`. If your tree has the most recent tip at time zero (i.e., the modern day), this would be the same as the root age of the tree. If your branch lengths are

	not in millions of years, you should alter this argument. Otherwise, leave this argument alone. See documentation for `doSimulation` for further details.
`timeStep`	This value corresponds to the length of intervals between discrete evolutionary events ('generations') simulated along branches, relative to a rescaled tree where the root to furthest tip distance is 1. For example, `timeStep = 0.01` of would mean 100 (i.e., 1 / 0.01) evolutionary changes would be expected to occur from the root to the furthest tip. (Note that the real number simulated will be much less, because simulations start over at each branching node.) Ideally, `timeStep` (or its effective value, via other arguments) should be as short as is computationally possible. Typically `NULL` by default and determined internally as follows: `timeStep = generation.time / TreeYears`. Can be provided a value as an alternative to using arguments `generation.time` and `TreeYears`, which would then be overridden. See documentation for `doSimulation` for further details.
`giveUpAttempts`	Value for when to stop the analysis if `NA` values are present.
`verbose`	If `TRUE`, gives messages about how the simulation is progessing via `message`.
`checks`	If `TRUE`, checks inputs for consistency. This activity is skipped (`checks = FALSE`) when run in parallel by `parallelSimulateWithPriors`, and instead is only checked once. This argument also controls whether `simulateWithPriors` assigns `freevector` as an attribute to the output produced.
`taxonDF`	A data.frame containing data on nodes (both tips and internal nodes) output by various internal functions. Can be supplied as input to spead up repeated calculations, but by default is `NULL`, which instead forces a calculation from input `phy`.
`freevector`	A logical vector (with length equal to the number of parameters), indicating free (`TRUE`) and fixed (`FALSE`) parameters.
`nrepSim`	Number of replicated simulations to run.
`multicore`	Whether to use multicore, default is `FALSE`. If `TRUE`, one of two suggested packages must be installed, either `doMC` (for UNIX systems) or `doParallel` (for Windows), which are used to activate multithreading. If neither package is installed, this function will fail if `multicore = TRUE`.
`coreLimit`	Maximum number of cores to be used.
`checkpointFile`	Optional file name for checkpointing simulations
`checkpointFreq`	Saving frequency for checkpointing
`checkTimeStep`	If `TRUE`, warnings will be issued if `TimeStep` is too short.
`verboseNested`	Should looped runs of `simulateWithPriors` be verbose?

Value

Function `simulateWithPriors` returns a vector of `trueFreeValues`, the true generating parameters used in the simulation (a set of values as long as the number of freely varying parameters), concatenated with a set of summary statistics for the simulation.

Function `parallelSimulateWithPriors` returns a matrix of such vectors bound together, with each row representing a different simulation.

By default, both functions also assign a logical vector named `freevector`, indicating the total number of parameters and which parameters are freely-varying (have `TRUE` values), as an attribute of the output.

Note

The simulateWithPriors functions are effectively the engine that powers the doRun functions, while the doSimulation function is the pistons within the simulateWithPriors engine. In general, most users will just drive the car - they will just use doRun, but some users may want to use simulateWithPriors or doSimulation to do various simulations.

Author(s)

Brian O'Meara and Barb Banbury

Examples

```
set.seed(1)
tree <- rcoal(20)
# get realistic edge lengths
tree$edge.length <- tree$edge.length*20

# example simulation

# NOTE: the example analyses involve too few simulations,
    # as well as overly coarse time-units...
    # ...all for the sake of examples that reasonably test the functions

simData <- simulateWithPriors(
  phy = tree,
  intrinsicFn = brownianIntrinsic,
  extrinsicFn = nullExtrinsic,
  startingPriorsFns = "normal",
  startingPriorsValues = list(
      c(mean(simCharExample[, 1]), sd(simCharExample[, 1]))),
  intrinsicPriorsFns = c("exponential"),
  intrinsicPriorsValues = list(10),
  extrinsicPriorsFns = c("fixed"),
  extrinsicPriorsValues = list(0),
  generation.time = 100000,
  freevector = NULL,
  giveUpAttempts = 10,
  verbose = TRUE)

simData

simDataParallel <- parallelSimulateWithPriors(
  nrepSim = 2,
  multicore = FALSE,
  coreLimit = 1,
  phy = tree,
  intrinsicFn = brownianIntrinsic,
  extrinsicFn = nullExtrinsic,
  startingPriorsFns = "normal",
  startingPriorsValues = list(
     c(mean(simCharExample[, 1]), sd(simCharExample[, 1]))),
  intrinsicPriorsFns = c("exponential"),
  intrinsicPriorsValues = list(10),
  extrinsicPriorsFns = c("fixed"),
  extrinsicPriorsValues = list(0),
  generation.time = 100000,
  checkpointFile = NULL, checkpointFreq = 24,
  verbose = TRUE,
  freevector = NULL,
  taxonDF = NULL)

simDataParallel
```

---

summarizePosterior *Summarize Posterior Distribution for a Free Parameter*

Description

This function summarizes the distribution of parameter estimates from the posterior of an ABC analysis using the doRun functions in `TreEvo`, for all freely varying parameters. Only the final generation is considered. This summary includes the mean, standard deviation and Highest Posterior Density (at a 0.8 alpha) for each parameter.

Usage

```
summarizePosterior(
  particleDataFrame,
  alpha = 0.8,
  coda = FALSE,
  verboseMultimodal = TRUE,
  stopIfFlat = TRUE,
  ...
)
```

Arguments

`particleDataFrame`	A `particleDataFrame` object, as found among the output from `doRun` functions.
`alpha`	The threshold used for defining the highest density frequency cut-off. If the highest density interval is applied to a Bayesian MCMC posterior sample, then the interval is effectively calculated for this value as a posterior probability density.
`coda`	Default is `FALSE`. If `TRUE`, unimodal highest density regions will instead be calculated using `HPDinterval` from package `coda`, which is similar to the function `quantile` in that it calculates only a single interval.

`verboseMultimodal`
: If `TRUE`, the function will print a message indicating when the inferred highest density interval is discontinuous, and thus likely reflects that the supplied data is multimodal.

`stopIfFlat`
: If `TRUE` (the default), the function will terminate and return an error if the distribution appears to be flat, with a high number of tied values in the posterior. If `FALSE`, the function will instead attempt to move forward, while returning a warning instead. The latter option is mainly implemented so internal calls to `highestDensityInterval` do not block otherwise long-running analyses.

`...`
: Additional arguments passed to `density`. A user may want to mess with this to adjust bandwidth, et cetera.

Value

Returns a list, wherein each element of the list is secondary list containing the weighted mean, standard deviation, and a matrix giving the highest density intervals (e.g. the highest posterior density intervals). Because posterior estimates of parameter values may be multimodal, multiple sets of bounds may be reported for complex posterior distributions, which each constitute one row of the output matrix. See `highestDensityInterval` for details.

Author(s)

David W Bapst

See Also

This function is essentially a wrapper for independently applying a few summary statistics and applying `highestDensityInterval` to multiple parameter estimates, taken from the last generation of an ABC analysis in `TreEvo`. As each parameter is handled independently, the returned HPD intervals may not properly account for covariation among parameter estimates from the posterior. If testing whether a given observation is within a given density of the posterior or not, please look at function `testMultivarOutlierHDR`.

Examples

```
# example with output from doRun_prc
data(simRunExample)

# alpha = 0.95
summarizePosterior(
     resultsBMExample[[1]]$particleDataFrame,
     alpha = 0.95)

# you might be tempted to use alphas like 95%,
    # but with bayesian statistics
# we often don't sample the distribution well enough to know
    # its shape to exceeding detail.
    # alpha = 0.8 may be more reasonable.
summarizePosterior(
    resultsBMExample[[1]]$particleDataFrame,
    alpha = 0.8)

# or even better, for coverage purposes, maybe 0.5
summarizePosterior(
    resultsBMExample[[1]]$particleDataFrame,
    alpha = 0.5)
```

---

testMultivarOutlierHDR

Test If an Outlier Is Within a Highest Density Region of a Multivariate Cloud of Data Points

---

Description

This function takes a matrix, consisting of multiple observations for a set of variables, with observations assumed to be independent and identically distributed, calculates a highest density interval for each of those variables (using `highestDensityInterval`), and then tests if some potential 'outlier' (a particular observation for that set of variables), is within that highest density interval. Typically, users may want to account for possible non-independence of the variables, which could lead to inflated false-positive rates with this test of coverage. To account for this, this function allows for the option of applying principal component analysis to rotate the variables, and then calculate the highest density intervals from the orthogonal principal component scores.

Usage

```
testMultivarOutlierHDR(
  dataMatrix,
  outlier,
  alpha,
  coda = FALSE,
  verboseMultimodal = FALSE,
  pca = TRUE,
  ...
)
```

Arguments

`dataMatrix`	A matrix consisting of rows of data observations, with one or more variables as the columns, such that each multivariate observation can be reasonably assumed to represent independent, identically-distributed random variables. For example, a matrix of sampled parameter estimates from the post-burnin posterior of a Bayesian MCMC.
`outlier`	A vector of consisting of a single observation of one or more variables, with the same length as the number of columns in /codedataMatrix. Not necessarily a true 'outlier' *per se* but some data point of interest that you wish to test whether it is inside some given probability density estimated from a sample of points.

`alpha`	The threshold used for defining the highest density frequency cut-off. If the highest density interval is applied to a Bayesian MCMC posterior sample, then the interval is effectively calculated for this value as a posterior probability density.
`coda`	Default is `FALSE`. If `TRUE`, unimodal highest density regions will instead be calculated using `HPDinterval` from package `coda`, which is similar to the function `quantile` in that it calculates only a single interval.
`verboseMultimodal`	If `TRUE`, the function will print a message indicating when the inferred highest density interval is discontinuous, and thus likely reflects that the supplied data is multimodal.
`pca`	If `TRUE` (the default), a principal components analysis is applied to the provided input data, to reorient the data along orthogonal axes, with the purpose of potentially reducing covariation among the variables. If your variables are known to have little covariation among them, and/or have strange distributions that may violate the multivariate normal assumptions of a principal components analysis, then it may be advisable to set pca = FALSE.
`...`	Additional arguments passed to `density`. A user may want to mess with this to adjust bandwidth, et cetera.

Details

Quantiles, or highest density intervals are essentially a univariate concept. They are calculated independently on each variable, as if each variable was independent. They are often used to try to piece together 'regions' of confidence or credibility in statistics. Unfortunately, this means that that if variables are correlated, the box-like region they describe in the multivariate space may not cover enough of the actual data scatter, while covering lots of empty space.

There are several solutions. There are a number of approaches for fitting ellipsoids to bivariate data. But what about multivariate datasets, such as sets of parameter estimates from the posterior of a Bayesian analysis, which may have an arbitrary number of variables (and thus dimensions)?

A different solution, applied here when `pca = TRUE`, is to remove the non-independence among variables by rotating the dataset with principal components analysis. While this approach has its own flaws, such as assuming that the data reflects a multivariate normal. However, this is absolutely an improvement on not-rotating, particularly if being done for the purpose of this functon, which is essentially to measure coverage–to measure whether some particular set of values falls within a highest density region (a multivariate highest density interval).

Value

A logical, indicating whether the given data point (`outlier`) is within the multivariate data cloud at the specified threshold probability density.

Author(s)

David W. Bapst

See Also

This function is essentially a wrapper for applying `highestDensityInterval` to multivariate data, along with `princomp` in package `stats`.

Examples

```
# First, you should understand the problems
# with dealing with correlated variables and looking at quantiles

# simulate two correlated variables
set.seed(444)
x <- rnorm(100, 1, 1)
y <- (x*1.5)+rnorm(100)

# find the highest density intervals for each variable
pIntX <- highestDensityInterval(x, alpha = 0.8)
pIntY <- highestDensityInterval(y, alpha = 0.8)

# These define a box-like region that poorly
# describes the actual distribution of
# the data in multivariate space.

# Let's see this ourselves...
xLims <- c(min(c(x, pIntX)), max(c(x, pIntX)))
yLims <- c(min(c(y, pIntY)), max(c(y, pIntY)))
plot(x, y, xlim = xLims, ylim = yLims)
rect(pIntX[1], pIntY[1], pIntX[2], pIntY[2])

# So, that doesn't look good.
# Let's imagine we wanted to test if some outlier
# was within that box:

outlier <- c(2, -1)
points(x = outlier[1], y = outlier[2], col = 2)

# we can use testMultivarOutlierHDR with pca = FALSE
# to do all of the above without visually checking
testMultivarOutlierHDR(dataMatrix = cbind(x, y),
    outlier = outlier, alpha = 0.8, pca = FALSE)

# Should that really be considered to be within
# the 80% density region of this data cloud?

#####

# let's try it with PCA

pcaRes <- princomp(cbind(x, y))
PC <- pcaRes$scores

pIntPC1 <- highestDensityInterval(PC[, 1], alpha = 0.8)
pIntPC2 <- highestDensityInterval(PC[, 2], alpha = 0.8)
```

```
# plot it
xLims <- c(min(c(PC[, 1], pIntPC1)), max(c(PC[, 1], pIntPC1)))
yLims <- c(min(c(PC[, 2], pIntPC2)), max(c(PC[, 2], pIntPC2)))
plot(PC[, 1], PC[, 2], xlim = xLims, ylim = yLims)
rect(pIntPC1[1], pIntPC2[1], pIntPC1[2], pIntPC2[2])

# That looks a lot better, isnt' filled with lots of
# white space not supported by the observed data.

# And now let's apply testMultivarOutlierHDR, with pca = TRUE
testMultivarOutlierHDR(dataMatrix = cbind(x, y),
    outlier = outlier, alpha = 0.8, pca = TRUE)

#####################

# Example with four complex variables
    # including correlated and multimodal variables

x <- rnorm(100, 1, 1)
y <- (x*0.8)+rnorm(100)
z <- sample(c(rnorm(50, 3, 2),
  rnorm(50, 30, 3)))
a <- sample(c(rnorm(50, 3, 2),
  rnorm(50, 10, 3)))+x^2

#plot(x, y)
#plot(x, z)
#plot(x, a)
data <- cbind(x, y, z, a)

# actual outlier, but maybe not obvious if PCA isn't applied
outlier <- c(2, 0.6, 10, 8)
# this one should appear to be an outlier (but only if PCA is applied)
testMultivarOutlierHDR(dataMatrix = data,
  outlier = outlier, alpha = 0.8)
testMultivarOutlierHDR(dataMatrix = data,
  outlier = outlier, alpha = 0.8, pca = FALSE)

# this one should be within the 80% area
outlier <- c(1, 0, 30, 5)
testMultivarOutlierHDR(dataMatrix = data,
  outlier = outlier, alpha = 0.8)
testMultivarOutlierHDR(dataMatrix = data,
  outlier = outlier, alpha = 0.8, pca = FALSE)

# this one should be an obvious outlier no matter what
outlier <- c(3, -2, 20, 18)
# this one should be outside the 80% area
testMultivarOutlierHDR(dataMatrix = data,
  outlier = outlier, alpha = 0.8)
testMultivarOutlierHDR(dataMatrix = data,
  outlier = outlier, alpha = 0.8, pca = FALSE)
```

Index