Package 'greta.dynamics'

November 14, 2024

Type Package

Title Modelling Structured Dynamical Systems in 'greta'

Version 0.2.2

Description A 'greta' extension for analysing transition matrices and ordinary differential equations representing dynamical systems. Provides functions for analysing transition matrices by iteration, and solving ordinary differential equations. This is an extension to the 'greta' software, Golding (2019) <doi:10.21105/joss.01601>.

License Apache License (>= 2)

URL https://github.com/greta-dev/greta.dynamics, https://greta-dev.github.io/greta.dynamics/

BugReports https://github.com/greta-dev/greta.dynamics/issues

Imports cli (>= 3.6.3), glue, rlang (>= 1.1.4), tensorflow (>= 1.14.0)

Depends greta (>= 0.5.0), R (>= 4.1.0)

Suggests covr, knitr, rmarkdown, spelling, testthat (>= 3.1.0), deSolve, abind

VignetteBuilder knitr

Config/testthat/edition 3

Encoding UTF-8

Language en-GB

RoxygenNote 7.3.2

SystemRequirements Python (>= 3.7.0) with header files and shared library; TensorFlow (>= v2.0.0; https://www.tensorflow.org/); TensorFlow Probability (v0.8.0; https://www.tensorflow.org/probability/)

NeedsCompilation no

Author Nick Golding [aut, cph] (<https://orcid.org/0000-0001-8916-5570>), Nicholas Tierney [aut, cre] (<https://orcid.org/0000-0003-1460-8722>)

Maintainer Nicholas Tierney <nicholas.tierney@gmail.com>

Repository CRAN

Date/Publication 2024-11-14 04:50:02 UTC

Contents

`greta.dynamics` *greta.dynamics: a greta extension for modelling dynamical systems*

Description

an extension to greta with functions for simulating dynamical systems, defined by of ordinary differential equations (see `ode_solve()`) or transition matrices (`iterate_matrix()`).

Author(s)

Maintainer: Nicholas Tierney <nicholas.tierney@gmail.com> (ORCID)

Authors:

- Nick Golding <nick.golding.research@gmail.com> (ORCID) [copyright holder]

See Also

Useful links:

- https://github.com/greta-dev/greta.dynamics
- https://greta-dev.github.io/greta.dynamics/
- Report bugs at https://github.com/greta-dev/greta.dynamics/issues

---

`iterate_dynamic_function` *iterate dynamic transition functions*

Description

Calculate the stable population size for a stage-structured dynamical system, encoded by a transition function, the value of which changes at each iteration, given by function of the previous state: `state[t] = f(state[t-1])`.

Usage

```
iterate_dynamic_function(
  transition_function,
  initial_state,
  niter,
  tol,
  ...,
  parameter_is_time_varying = c(),
  state_limits = c(-Inf, Inf)
)
```

Arguments

`transition_function`	a function taking in the previous population state and the current iteration (and possibly other greta arrays) and returning the population state at the next iteration. The first two arguments must be named 'state' and 'iter', the state vector and scalar iteration number respectively. The remaining parameters must be named arguments representing (temporally static) model parameters. Variables and distributions cannot be defined inside the function.
`initial_state`	either a column vector (with m elements) or a 3D array (with dimensions n x m x 1) giving one or more initial states from which to iterate the matrix
`niter`	a positive integer giving the maximum number of times to iterate the matrix
`tol`	a scalar giving a numerical tolerance, below which the algorithm is determined to have converged to a stable population size in all stages
`...`	optional named arguments to `matrix_function`, giving greta arrays for additional parameters
`parameter_is_time_varying`	a character vector naming the parameters (ie. the named arguments of the function that are passed via `...`) that should be considered to be time-varying. That is, at each iteration only the corresponding slice from the first dimension of the object passed in should be used at that iteration.
`state_limits`	a numeric vector of length 2 giving minimum and maximum values at which to clamp the values of state after each iteration to prevent numerical under/overflow; i.e. elements with values below the minimum (maximum) will be set to the minimum (maximum).

Details

Like `iterate_dynamic_matrix` this converges to *absolute* population sizes. The convergence criterion is therefore based on growth rates converging on 0.

The greta array returned by `transition_function` must have the same dimension as the state input and `initial_state` should be shaped accordingly, as detailed in `iterate_matrix`.

To ensure the matrix is iterated for a specific number of iterations, you can set that number as `niter`, and set `tol` to 0 or a negative number to ensure that the iterations are not stopped early.

Value

a named list with four greta arrays:

- `stable_state` a vector or matrix (with the same dimensions as `initial_state`) giving the state after the final iteration.
- `all_states` an n x m x niter matrix of the state values at each iteration. This will be 0 for all entries after `iterations`.
- `converged` an integer scalar indicating whether *all* the matrix iterations converged to a tolerance less than `tol` (1 if so, 0 if not) before the algorithm finished.
- `iterations` a scalar of the maximum number of iterations completed before the algorithm terminated. This should match `niter` if `converged` is `FALSE`

Note

because greta vectorises across both MCMC chains and the calculation of greta array values, the algorithm is run until all chains (or posterior samples), sites and stages have converged to stable growth. So a single value of both `converged` and `iterations` is returned, and the value of this will always have the same value in an `mcmc.list` object. So inspecting the MCMC trace of these parameters will only tell you whether the iteration converged in *all* posterior samples, and the maximum number of iterations required to do so across all these samples

---

`iterate_dynamic_matrix` *iterate dynamic transition matrices*

Description

Calculate the stable population size for a stage-structured dynamical system, encoded by a transition matrix, the value of which changes at each iteration, given by function of the previous state: `state[t] = f(state[t-1]) %*% state[t-1]`.

Usage

```
iterate_dynamic_matrix(
  matrix_function,
  initial_state,
  niter,
  tol,
  ...,
  state_limits = c(-Inf, Inf)
)
```

Arguments

`matrix_function`	a function taking in the previous population state and the current iteration (and possibly other greta arrays) and returning a transition matrix to use for this iteration. The first two arguments must be named 'state' and 'iter', the state vector and scalar iteration number respectively. The remaining parameters must be named arguments representing (temporally static) model parameters. Variables and distributions cannot be defined inside the function.
`initial_state`	either a column vector (with m elements) or a 3D array (with dimensions n x m x 1) giving one or more initial states from which to iterate the matrix
`niter`	a positive integer giving the maximum number of times to iterate the matrix
`tol`	a scalar giving a numerical tolerance, below which the algorithm is determined to have converged to a stable population size in all stages
`...`	optional named arguments to `matrix_function`, giving greta arrays for additional parameters
`state_limits`	a numeric vector of length 2 giving minimum and maximum values at which to clamp the values of state after each iteration to prevent numerical under/overflow; i.e. elements with values below the minimum (maximum) will be set to the minimum (maximum).

Details

Because `iterate_matrix` iterates with a static transition matrix, it converges to a stable *growth rate* and *relative* population sizes for a dynamical system. `iterate_dynamic_matrix` instead uses a matrix which changes at each iteration, and can be dependent on the population sizes after the previous iteration, and the iteration number. Because this can encode density-dependence, the dynamics can converge to *absolute* population sizes. The convergence criterion is therefore based on growth rates converging on 0.

As in `iterate_matrix`, the greta array returned by `matrix_function` can either be a square matrix, or a 3D array representing (on the first dimension) *n* different matrices. `initial_state` should be shaped accordingly, as detailed in `iterate_matrix`.

To ensure the matrix is iterated for a specific number of iterations, you can set that number as `niter`, and set `tol` to 0 or a negative number to ensure that the iterations are not stopped early.

Value

a named list with four greta arrays:

- `stable_state` a vector or matrix (with the same dimensions as `initial_state`) giving the state after the final iteration.
- `all_states` an n x m x niter matrix of the state values at each iteration. This will be 0 for all entries after `iterations`.
- `converged` an integer scalar indicating whether *all* the matrix iterations converged to a tolerance less than `tol` (1 if so, 0 if not) before the algorithm finished.
- `iterations` a scalar of the maximum number of iterations completed before the algorithm terminated. This should match `niter` if `converged` is `FALSE`

Note

because greta vectorises across both MCMC chains and the calculation of greta array values, the algorithm is run until all chains (or posterior samples), sites and stages have converged to stable growth. So a single value of both `converged` and `iterations` is returned, and the value of this will always have the same value in an `mcmc.list` object. So inspecting the MCMC trace of these parameters will only tell you whether the iteration converged in *all* posterior samples, and the maximum number of iterations required to do so across all these samples

---

`iterate_matrix` *iterate transition matrices*

Description

Calculate the intrinsic growth rate(s) and stable stage distribution(s) for a stage-structured dynamical system, encoded as `state_t = matrix \%*\% state_tm1`.

Usage

```
iterate_matrix(
  matrix,
  initial_state = rep(1, ncol(matrix)),
  niter = 100,
  tol = 1e-06
)
```

Arguments

`matrix`	either a square 2D transition matrix (with dimensions m x m), or a 3D array (with dimensions n x m x m), giving one or more transition matrices to iterate
`initial_state`	either a column vector (with m elements) or a 3D array (with dimensions n x m x 1) giving one or more initial states from which to iterate the matrix
`niter`	a positive integer giving the maximum number of times to iterate the matrix
`tol`	a scalar giving a numerical tolerance, below which the algorithm is determined to have converged to the same growth rate in all stages

Details

`iterate_matrix` can either act on a single transition matrix and initial state (if `matrix` is 2D and `initial_state` is a column vector), or it can simultaneously act on *n* different matrices and/or *n* different initial states (if `matrix` and `initial_state` are 3D arrays). In the latter case, the first dimension of both objects should be the batch dimension *n*.

To ensure the matrix is iterated for a specific number of iterations, you can set that number as `niter`, and set `tol` to 0 or a negative number to ensure that the iterations are not stopped early.

Value

a named list with five greta arrays:

- `lambda` a scalar or vector giving the ratio of the first stage values between the final two iterations.
- `stable_state` a vector or matrix (with the same dimensions as `initial_state`) giving the state after the final iteration, normalised so that the values for all stages sum to one.
- `all_states` an n x m x niter matrix of the state values at each iteration. This will be 0 for all entries after `iterations`.
- `converged` an integer scalar or vector indicating whether the iterations for each matrix have converged to a tolerance less than `tol` (1 if so, 0 if not) before the algorithm finished.
- `iterations` a scalar of the maximum number of iterations completed before the algorithm terminated. This should match `niter` if `converged` is `FALSE`.

Note

because greta vectorises across both MCMC chains and the calculation of greta array values, the algorithm is run until all chains (or posterior samples), sites and stages have converged to stable growth. So a single value of both `converged` and `iterations` is returned, and the value of this will always have the same value in an `mcmc.list` object. So inspecting the MCMC trace of these parameters will only tell you whether the iteration converged in *all* posterior samples, and the maximum number of iterations required to do so across all these samples

Examples

```
## Not run:
# simulate from a probabilistic 4-stage transition matrix model
k <- 4

# component variables
# survival probability for all stages
survival <- uniform(0, 1, dim = k)
# conditional (on survival) probability of staying in a stage
stasis <- c(uniform(0, 1, dim = k - 1), 1)
# marginal probability of staying/progressing
stay <- survival * stasis
progress <- (survival * (1 - stay))[1:(k - 1)]
# recruitment rate for the largest two stages
recruit <- exponential(c(3, 5))

# combine into a matrix:
tmat <- zeros(k, k)
diag(tmat) <- stay
progress_idx <- row(tmat) - col(tmat) == 1
tmat[progress_idx] <- progress
tmat[1, k - (1:0)] <- recruit

# analyse this to get the intrinsic growth rate and stable state
iterations <- iterate_matrix(tmat)
iterations$lambda
```

```
iterations$stable_distribution
iterations$all_states

# Can also do this simultaneously for a collection of transition matrices
k <- 2
n <- 10
survival <- uniform(0, 1, dim = c(n, k))
stasis <- cbind(uniform(0, 1, dim = n), rep(1, n))
stay <- survival * stasis
progress <- (survival * (1 - stasis))[, 1]
recruit_rate <- 1 / seq(0.1, 5, length.out = n)
recruit <- exponential(recruit_rate, dim = n)
tmats <- zeros(10, 2, 2)
tmats[, 1, 1] <- stasis[, 1]
tmats[, 2, 2] <- stasis[, 2]
tmats[, 2, 1] <- progress
tmats[, 1, 2] <- recruit

iterations <- iterate_matrix(tmats)
iterations$lambda
iterations$stable_distribution
iterations$all_states

## End(Not run)
```

`ode_solve` *solve ODEs*

Description

Solve a system of ordinary differential equations.

Usage

```
ode_solve(derivative, y0, times, ..., method = c("dp", "bdf"))
```

Arguments

`derivative`	a derivative function. The first two arguments must be 'y' and 't', the state parameter and scalar timestep respectively. The remaining parameters must be named arguments representing (temporally static) model parameters. Variables and distributions cannot be defined in the function.
`y0`	a greta array for the value of the state parameter y at time 0
`times`	a column vector of times at which to evaluate y
`...`	named arguments giving greta arrays for the additional (fixed) parameters
`method`	which solver to use. Default is "dp", which is similar to deSolve's "ode45". Currently implemented is "dp", and "bdf".The "dp" solver is Dormand-Prince explicit solver for non-stiff ODEs. The "bdf" solver is Backward Differentiation Formula (BDF) solver for stiff ODEs. Currently no arguments for "bdf" or "dp" are able to be specified.

Value

greta array

Examples

```
## Not run:
# replicate the Lotka-Volterra example from deSolve
library(deSolve)
LVmod <- function(Time, State, Pars) {
  with(as.list(c(State, Pars)), {
    Ingestion <- rIng * Prey * Predator
    GrowthPrey <- rGrow * Prey * (1 - Prey / K)
    MortPredator <- rMort * Predator

    dPrey <- GrowthPrey - Ingestion
    dPredator <- Ingestion * assEff - MortPredator

    return(list(c(dPrey, dPredator)))
  })
}

pars <- c(
  rIng = 0.2, # /day, rate of ingestion
  rGrow = 1.0, # /day, growth rate of prey
  rMort = 0.2, # /day, mortality rate of predator
  assEff = 0.5, # -, assimilation efficiency
  K = 10
) # mmol/m3, carrying capacity

yini <- c(Prey = 1, Predator = 2)
times <- seq(0, 30, by = 1)
out <- ode(yini, times, LVmod, pars)

# simulate observations
jitter <- rnorm(2 * length(times), 0, 0.1)
y_obs <- out[, -1] + matrix(jitter, ncol = 2)

# ~~~~~~~~~
# fit a greta model to infer the parameters from this simulated data

# greta version of the function
lotka_volterra <- function(y, t, rIng, rGrow, rMort, assEff, K) {
  Prey <- y[1, 1]
  Predator <- y[1, 2]

  Ingestion <- rIng * Prey * Predator
  GrowthPrey <- rGrow * Prey * (1 - Prey / K)
  MortPredator <- rMort * Predator

  dPrey <- GrowthPrey - Ingestion
  dPredator <- Ingestion * assEff - MortPredator
```

```
  cbind(dPrey, dPredator)
}

# priors for the parameters
rIng <- uniform(0, 2) # /day, rate of ingestion
rGrow <- uniform(0, 3) # /day, growth rate of prey
rMort <- uniform(0, 1) # /day, mortality rate of predator
assEff <- uniform(0, 1) # -, assimilation efficiency
K <- uniform(0, 30) # mmol/m3, carrying capacity

# initial values and observation error
y0 <- uniform(0, 5, dim = c(1, 2))
obs_sd <- uniform(0, 1)

# solution to the ODE
y <- ode_solve(lotka_volterra, y0, times, rIng, rGrow, rMort, assEff, K)

# sampling statement/observation model
distribution(y_obs) <- normal(y, obs_sd)

# we can use greta to solve directly, for a fixed set of parameters (the true
# ones in this case)
values <- c(
  list(y0 = t(1:2)),
  as.list(pars)
)
vals <- calculate(y, values = values)[[1]]
plot(vals[, 1] ~ times, type = "l", ylim = range(vals))
lines(vals[, 2] ~ times, lty = 2)
points(y_obs[, 1] ~ times)
points(y_obs[, 2] ~ times, pch = 2)

# or we can do inference on the parameters:

# build the model (takes a few seconds to define the tensorflow graph)
m <- model(rIng, rGrow, rMort, assEff, K, obs_sd)

# compute MAP estimate
o <- opt(m)
o

## End(Not run)
```

Index