Package 'tdsa'

May 15, 2025

Version 1.1-1

Date 2025-05-15

Title Time-Dependent Sensitivity Analysis

Depends R (>= 3.5.0)

Imports deSolve (>= 1.10-6), mathjaxr (>= 0.8-3), numDeriv (>= 2006.4-1)

Suggests knitr, tinytest

RdMacros mathjaxr

VignetteBuilder knitr

Description Functions that can be used to calculate time-dependent state and parameter sensitivities for both continuous- and discrete-time deterministic models. See Ng et al. (2023) <doi:10.1086/726143> for more information about time-dependent sensitivity analysis.

License GPL-3

URL https://github.com/weehaong/tdsa

BugReports https://github.com/weehaong/tdsa/issues

R topics documented:

parm_sens 1
state_sens 4

Index **11**

`parm_sens` *Time-Dependent Parameter Sensitivities*

Description

Function to calculate time-dependent parameter sensitivities.

Assume the same model and reward as described in `state_sens`. Unlike perturbations of the state variables, since the model parameters are not treated as dynamic quantities (even if they may be time-varying), an explicit perturbation of a parameter will only temporarily change the parameter while the perturbation lasts. Now consider a very brief perturbation (i.e., a sharp spike or dip) of the parameter b_i, centered at time t. We define the time-dependent parameter sensitivity $\kappa_i(t)$ as

the sensitivity of the reward to such a perturbation. See Ng et al. (in press, submitted) for a more precise definition.

This function uses the output returned by `state_sens` (which contain elements `parms` and `times`) to calculate the sensitivity for every parameter in `parms` at every time step in `times`.

See `state_sens` for examples.

Note: `parm_sens` assumes that the reward function does not depend explicitly on the parameters of interest, so any parameter perturbation will only affect the reward indirectly through its effects on the state variables. If this assumption is not true, then there is an additional 'direct' contribution that needs to be added to the results; we will show how this can be done in a future vignette.

Usage

```
parm_sens(
  state_sens_out,
  numDeriv_arglist = list(),
  verbose = TRUE
)
```

Arguments

`state_sens_out`	Output returned by `state_sens`. List containing the elements `model_type`, `dynamic_fn`, `parms`, `dynamic_fn_arglist`, `times`, `state` and `tdss`. To make this help page easier to read, from now on, any time we mention `dynamic_fn`, `parms`, etc., we refer to the corresponding elements in `state_sens_out`.
`numDeriv_arglist`	Optional list of arguments passed to the function `jacobian` from the **numDeriv** package, when calculating derivatives. Can be used to specify the method, and arguments controlling the method. For example, if the parameter sensitivities take too long to calculate, try setting `numDeriv_arglist = list(method="simple")` to replace Richardson's extrapolation by a simple one-sided epsilon difference.
`verbose`	Whether to display progress messages in the console. Either `TRUE` (the default) or `FALSE`.

Details

Parameter sensitivities can be obtained from the state sensitivities using the following formulae.

- **Continuous-time models:**

$$\kappa_i(t) = \sum_{j=1}^{n_y} \left. \frac{\partial g_j(t, \mathbf{y}(t), \mathbf{b})}{\partial b_i} \right|_{\mathbf{b}=\mathbf{b}(t)} \lambda_j(t),$$

  where $\lambda_j(t)$ is the state sensitivity of y_j at time t.

- **Discrete-time models:**

$$\kappa_i(t) = \sum_{j=1}^{n_y} \left. \frac{\partial g_j(t, \mathbf{y}(t), \mathbf{b})}{\partial b_i} \right|_{\mathbf{b}=\mathbf{b}(t)} \lambda_j(t+1),$$

  where $\lambda_j(t+1)$ is the state sensitivity of y_j at time step $t+1$. This also means that the parameter sensitivities are always zero at the final time step t_1, because $\lambda_j(t_1+1) = 0$ for all j.

To apply these formulae, we need to calculate derivatives of `dynamic_fn` with respect to `parms`, using the function `jacobian` from **numDeriv**. The main coding challenge that we have addressed is to make this work even when the structure of `parms` is only under the relatively mild restrictions imposed in `state_sens`.

Value

A list with the following elements:

`times` Time steps at which the parameter sensitivities are evaluated, a numeric vector. Same as `times` from `state_sens_out`.

`tdps` Time-dependent parameter sensitivities. An object whose structure depends on the structure of `parms`.

- If `parms` is a numeric object, then `tdps` is an array with one more index than the object, so a vector becomes a matrix, a matrix becomes a 3-index array, etc. The first index is new and is associated with the time step.
- If `parms` is a function of the form `function(t)` that returns a numeric object (i.e., time-varying parameters), then `tdps` is an array with one more index than the returned object. Again, the first index is new and is associated with the time step.
- If `parms` is a list containing any combination of numeric objects and functions, then `tdps` is a list of the same length, with the previous "rules" applied element-wise.

As a concrete example, say `parms` is a matrix of dimension `c(3,2)`, and `times` a vector of length `50`. Then `tdps` is a 3-index array of dimension `c(50,3,2)`, and the array element `tdps[20,1,2]` gives the sensitivity for the parameter `parms[1,2]` at time step `times[20]`.

Warning

The function `parm_sens` will calculate the sensitivities for **every** parameter in the argument `parms` used by `dynamic_fn`. Hence, `parms` should not contain discrete parameters such as the length of the vector or the dimensions of a matrix; otherwise `jacobian` (from the **numDeriv** package) will attempt to numerically evaluate the derivative of `dynamic_fn` with respect to such a discrete parameter and will hence almost invariably return an error message or nonsensical results. The solution is instead to make use the fact that we allow `dynamic_fn` to take additional arguments `...`; for example, we can define `dynamic_fn` to take an additional argument `parms2` that will then be used to hold these discrete parameters instead of `parms`.

References

Ng, W. H., Myers, C. R., McArt, S., & Ellner, S. P. (2023). A time for every purpose: using time-dependent sensitivity analysis to help understand and manage dynamic ecological systems. American Naturalist, 202, 630-654. doi: 10.1086/726143. eprint doi: 10.1101/2023.04.13.536769.

Ng, W. H., Myers, C. R., McArt, S., & Ellner, S. P. (2023). tdsa: An R package to perform time-dependent sensitivity analysis. Methods in Ecology and Evolution, 14, 2758-2765. doi: 10.1111/2041210X.14216.

See Also

`state_sens` for time-dependent state sensitivities.

state_sens *Time-Dependent State Sensitivities*

Description

Function to calculate time-dependent state sensitivities. Both continuous- and discrete-time models are supported.

- **Continuous-time models:** Assume that the dynamics of the system can be described by first-order ordinary differential equations (and initial conditions)

$$\frac{d\mathbf{y}(t)}{dt} = \mathbf{g}(t, \mathbf{y}(t), \mathbf{b}(t)), \quad \mathbf{y}(t_0) = \mathbf{y}_0,$$

  where $\mathbf{y}(t)$ is the n_y-dimensional state vector and $\mathbf{b}(t)$ the model parameters at time t. Also assume there is some reward of interest (e.g., a management objective) that can be written as

$$J = \int_{t_0}^{t_1} f(t, \mathbf{y}(t))\, dt + \Psi(\mathbf{y}(t_1)),$$

  where t_0 and t_1 are the initial and final times, and $\Psi(\mathbf{y}(t_1))$ represents a terminal payoff. (We will explain how to deal with non-standard objectives that cannot be expressed in such a form in a future vignette.)

- **Discrete-time models:** Choose the units of time so that the time steps take consecutive integer values. Assume that the dynamics of the system can be described by first-order recurrence equations (and initial conditions)

$$\mathbf{y}(t+1) = \mathbf{g}(t, \mathbf{y}(t), \mathbf{b}(t)), \quad \mathbf{y}(t_0) = \mathbf{y}_0.$$

  Also assume that the reward can be written as

$$J = \sum_{t=t_0}^{t_1-1} f(t, \mathbf{y}(t)) + \Psi(\mathbf{y}(t_1)).$$

We now consider a sudden perturbation of the ith state variable y_i at time t. Even though the perturbation only occurs explicitly at time t, it will "nudge" the system to a new state trajectory, so all state variables after time t will be affected. Hence the reward is affected by both the explicit perturbation as well as the "downstream" changes. We define the time-dependent state sensitivity $\lambda_i(t)$ as the sensitivity of the reward to such a perturbation; the sensitivity is time-dependent because it depends on the time t at which the perturbation occurs. See Ng et al. (in press, submitted) for a more precise definition.

This function calculates the sensitivity $\lambda_i(t)$ for every i from 1 to n_y, at every t between t_0 and t_1. Hence, the user can identify the state variable and the time of perturbation that would have the largest impact on the reward.

The output of this function can also be used as the input argument of the function parm_sens to calculate time-dependent parameter sensitivities.

Usage

```
state_sens(
  model_type,
  dynamic_fn,
  parms,
  reward_fn,
  terminal_fn,
  y_0,
  times,
  interpol = "spline",
  dynamic_fn_arglist = list(),
  reward_fn_arglist = list(),
  terminal_fn_arglist = list(),
  state_ode_arglist = list(),
  adjoint_ode_arglist = list(),
  numDeriv_arglist = list(),
  verbose = TRUE
)
```

Arguments

`model_type` Whether the model is continuous- or discrete-time. Allowed values are `"continuous"` and `"discrete"`.

`dynamic_fn` Dynamic equations of the state variables. Function of the form `function(t,y,parms,...)`, with arguments

`t` Time t, a single number.

`y` State vector $\mathbf{y}$, a numeric vector of length n_y.

`parms` Object used to specify the model parameters $\mathbf{b}(t)$. Allowed structures are:

- A numeric object. This can be a vector, matrix or array.
- A function of the form `function(t)`, that returns a numeric object. This is used for time-varying parameters. See "Details".
- A list containing any combination of the above.
- `NULL` if the user prefers to specify parameter values elsewhere.

We have imposed these restrictions to facilitate parameter sensitivity calculations using `parm_sens`, but nonetheless they should be mild enough to permit most use cases. See "Details."

`...` Additional arguments.

Function must return a list, whose first element is $\mathbf{g}(t, \mathbf{y}, \mathbf{b}(t))$, a numeric vector of length n_y. Other elements of the returned list are optional, and correspond to additional numeric quantities that the user wants to monitor at each time step.

Note to users of the **deSolve** package: Any function that can be used as `func` in `ode` can be used as `dynamic_fn`, provided `parms` has one of the allowed structures described above.

`parms` Argument passed to `dynamic_fn`.

`reward_fn` Integrand (continuous-time model) or summand (discrete-time model) in reward function. Function of the form `function(t,y,...)`, with arguments

`t` Time t, a single number.

`y` State vector $\mathbf{y}$, a numeric vector of length n_y.

`...` Additional arguments.

Function must return $f(t, \mathbf{y})$, a single number.

`terminal_fn` Terminal payoff in reward function. Function of the form `function(y,...)`, with arguments

`y` State vector $\mathbf{y}$, a numeric vector of length n_y.

`...` Additional arguments.

Function must return $\Psi(\mathbf{y})$, a single number.

`y_0` Initial conditions of the dynamical system $\mathbf{y}_0$, a numeric vector of length n_y.

`times` Numeric vector containing the time steps at which the state variables and sensitivities will be evaluated. Must be in ascending order, and not contain duplicates. The first and last time steps must be t_0 and t_1.

For continuous-time models, this is the discretisation of the continuous interval between t_0 and t_1, so the smaller the step sizes, the more accurate the numerical results.

For discrete-time models, this must be a vector of consecutive integers, so t_0 and t_1 must themselves be integers.

`interpol` Only used for continuous-time models. Whether to perform spline or linear interpolation of the numerical solutions of the state variables. Allowed values are `"spline"` (the default) and `"linear"`. The former uses the function `splinefun`, while the latter uses the function `approxfun`, both from the **stats** package.

`dynamic_fn_arglist, reward_fn_arglist, terminal_fn_arglist`

Optional lists of arguments passed to `dynamic_fn`, `reward_fn` and `terminal_fn`. Can be used to specify any additional arguments `...` that these functions were designed to accept.

`state_ode_arglist, adjoint_ode_arglist`

Only used for continuous-time models. Optional lists of arguments passed to the function `ode` from the **deSolve** package, when solving the dynamic and adjoint equations respectively. Can be used to specify the method, and arguments controlling the method. See "Details" for the definition of the adjoint equations. (Discrete-time models will always use the "iteration" method, so these arguments are ignored.)

`numDeriv_arglist`

Optional list of arguments passed to the functions `grad` and `jacobian` from the **numDeriv** package, when calculating derivatives. Can be used to specify the method, and arguments controlling the method. For example, if the adjoint equations take too long to solve, try setting `numDeriv_arglist = list(method="simple")` to replace Richardson's extrapolation by a simple one-sided epsilon difference.

`verbose` Whether to display progress messages in the console. Either `TRUE` (the default) or `FALSE`.

Details

Algorithm: This function uses the adjoint method to calculate the sensitivity for every state variable at every time step in `times`. It automates the following sequence of steps:

1. Obtain numerical solutions of the state variables at every time step, by solving the dynamic equations `dynamic_fn` forward in time using `ode` from **deSolve**, with initial conditions `y_0`. (Note that `ode` can also support discrete-time models using the "iteration" method.)
2. For continuous-time models, create a function that interpolates the numerical solutions of the state variables, using either `splinefun` or `approxfun` from **stats**. This step is not required for discrete-time models.

3. Define a function (internally called `adjoint_fn`) that returns the RHS of the adjoint equations.
   - **Continuous-time models:** The adjoint equations are the first-order ordinary differential equations

$$\frac{d\lambda_i(t)}{dt} = -\left.\frac{\partial f(t,\mathbf{y})}{\partial y_i}\right|_{\mathbf{y}=\mathbf{y}(t)} - \sum_j \lambda_j(t)\left.\frac{\partial g_j(t,\mathbf{y})}{\partial y_i}\right|_{\mathbf{y}=\mathbf{y}(t)}.$$

   - **Discrete-time models:** The adjoint equations are the first-order recurrence equations

$$\lambda_i(t-1) = \left.\frac{\partial f(t-1,\mathbf{y})}{\partial y_i}\right|_{\mathbf{y}=\mathbf{y}(t-1)} + \sum_j \lambda_j(t)\left.\frac{\partial g_j(t-1,\mathbf{y})}{\partial y_i}\right|_{\mathbf{y}=\mathbf{y}(t-1)}.$$

   Inside `adjoint_fn`, we use `jacobian` and `grad` from **numDeriv** to evaluate the Jacobian and gradient of `dynamic_fn` and `reward_fn`. For discrete-time models, the values of the state variables (at which these derivatives are evaluated) come directly from the numerical solutions from Step 1. For continuous-time model, ODE solvers need `adjoint_fn` to work at any time t and not just those in `times`, so the values of the state variables instead come from the interpolation function from Step 2.
4. Calculate the terminal conditions of the adjoint system

$$\lambda_i(t_1) = \left.\frac{\partial \Psi(\mathbf{y})}{\partial y_i}\right|_{\mathbf{y}=\mathbf{y}(t_1)},$$

   using `grad` to evaluate the gradient of `terminal_fn`.
5. Obtain numerical solutions of the adjoint variables, by solving the adjoint equations backward in time using `ode`, with the terminal conditions from Step 4. The values of the adjoint variables are equal to the time-dependent state sensitivities.

Parameters in `dynamic_fn`: As mentioned earlier, the output of `state_sens` can be used as the input argument of the function `parm_sens` to calculate parameter sensitivities. The following points are important if the user wants to do so, and can be ignored otherwise.

- There are four ways to specify parameters in `dynamic_fn`: (1) using `parms`, (2) using the additional arguments `...`, (3) within the environment of `dynamic_fn` itself, and (4) in the global environment. The function `parm_sens` will calculate sensitivities for **all** the parameters specified using (1), and none of the parameters specified using (2), (3) or (4). These calculations involve taking numerical derivatives of `dynamic_fn` with respect to the parameters, which is why we have imposed some (relatively mild) restrictions on the structure of `parms`.
- The usual way to implement time-varying parameters is to have `parms` be a function of time (or a list containing such a function), which is then evaluated at `t` within `dynamic_fn` itself to return the current parameter values. When calculating parameter sensitivities, it is important that the evaluation be at `t` and not at a shifted time like `t-1`. This is because to us the user-specified `dynamic_fn` is a "black box", so there is no way we would know if `dynamic_fn` is using an evaluation like `parms(t-1)` to obtain the current parameter values instead of `parms(t)`.

Value

A list with the following elements:

`model_type`, `dynamic_fn`, `parms`, `dynamic_fn_arglist`, `times`
: Same as the input arguments. Included in the output because they are needed for parameter sensitivity calculations using `parm_sens`.

`state` Numerical solutions of the state variables evaluated at `times`. Matrix with as many rows as the length of `times`, and as many columns as n_y (and possibly more; see below). The `i`th row corresponds to $(y_1(t), y_2(t), ..., y_{n_y}(t))$, where t is the time step `times[i]`.

If there are additional numeric quantitities that the user wants to monitor at each time step (these are the optional elements in the list returned by `dynamic_fn`), they will appear as additional columns to the right.

Note to users of the **deSolve** package: `state` is the usual output returned by `ode`, except with the first column (corresponding to `times`) removed. This is for consistency with the output returned by `parm_sens`.

`tdss` Time-dependent state sensitivities evaluated at `times`. Matrix with as many rows as the length of `times`, and as many columns as n_y. The `i`th row corresponds to $(\lambda_1(t), \lambda_2(t), ..., \lambda_{n_y}(t))$, where t is the time step `times[i]`.

References

Ng, W. H., Myers, C. R., McArt, S., & Ellner, S. P. (2023). A time for every purpose: using time-dependent sensitivity analysis to help understand and manage dynamic ecological systems. American Naturalist, 202, 630-654. doi: 10.1086/726143. eprint doi: 10.1101/2023.04.13.536769.

Ng, W. H., Myers, C. R., McArt, S., & Ellner, S. P. (2023). tdsa: An R package to perform time-dependent sensitivity analysis. Methods in Ecology and Evolution, 14, 2758-2765. doi: 10.1111/2041210X.14216.

See Also

`parm_sens` for time-dependent parameter sensitivities.

Examples

```
# Load the TDSA package.
library(tdsa)



# We will consider an example involving the translocation of individuals into a
# sink habitat that is being restored.

# -----------
# Background.
# -----------
# Consider an organism in a sink habitat, where the per-capita loss rate
# (mortality and emigration combined) exceeds the per-capita unregulated birth
# rate, so the population is only maintained through immigration. However, the
# mortality rate is expected to decrease over time due to ongoing habitat
# restoration efforts, so the population should eventually become
# self-sustaining. The population dynamics is hence given by
#
#   dy(t)/dt = b*y(t)*(1 - a*y(t)) - mu(t)*y(t) + sigma,
#
# where y(t) is the population at time t, b the unregulated per-capita birth
# rate, a the coefficient for reproductive competition, mu(t) the time-varying
# per-capita loss rate, and sigma the immigration rate. We assume that mu(t)
# starts off above b (so it is a sink habitat), but decreases as a sigmoidal
# and eventually falls below b (so the population becomes self-sustaining).
```

```
#
#
# The organism provides an important ecosystem service. Over a management period
# from t_0 to t_1, we ascribe an economic value to the organism
#
# J = integrate(w y(t), lower=t_0, upper=t_1)
#
# Here, w is the per-capita rate at which the service is provided, so the
# integral gives the total value of the service accumulated over the period.
# However, we also want to ascribe value to maintaining a large population at
# the end of the management period, so the second term corresponds to a terminal
# payoff where v is the ascribed value per individual.
#
#
# Say we want to translocate individuals to the habitat to speed up the
# population recovery and increase the reward J. What is the best time to do so
# in order to maximise the increase in the reward? As early as possible? Or only
# when the loss rate has become low enough that the population can sustain
# itself? A one-off translocation causes a small, sudden increase in the
# population size, so it is useful to look at the time-dependent state
# sensitivity. Alternatively, we can interpret the translocation as a brief
# spike in the immigration rate sigma, so we can also look at the time-dependent
# parameter sensitivity of sigma.



# ------------------------------
# Preparing the input arguments.
# ------------------------------
# Parameter values for the dynamic equations.
parms = list(
  b = 1,                                          # Per-capita birth rate.
  a = 0.1,                                        # Competition coefficient.
  mu = function(t){0.5 + 1/(1 + exp((t-10)/2))},  # Per-capita loss rate.
  sigma = 0.2                                     # Immigration rate.
)

# Function that returns the dynamic equations.
dynamic_fn = function(t, y, parms){
  b = parms[["b"]]
  a = parms[["a"]]
  sigma = parms[["sigma"]]
  mu = parms[["mu"]](t)

  dy = b*y*(1- a*y) - mu*y + sigma
  return( list(dy) )
}

# Initial conditions.
y_0 = 0.37  # Approximate steady-state population before restoration efforts.

# Function that returns the reward integrand.
reward_fn = function(t, y){
  w = 1  # Per-capita rate at which the ecosystem service is provided.
  return( w * y )
}
```

```
# Function that returns the terminal payoff.
terminal_fn = function(y){
  v = 1.74  # Ascribed value per individual at the end of the period.
  return( v * y )
}

# Time steps over management period. We discretise it into 1001 time steps
# (so the step size is 0.03).
times = seq(0, 30, length.out=1001)



# ------------------------------------------------
# Calculating time-dependent state sensitivities.
# ------------------------------------------------
state_sens_out = state_sens(
  model_type = "continuous",
  dynamic_fn = dynamic_fn,
  parms = parms,
  reward_fn = reward_fn,
  terminal_fn = terminal_fn,
  y_0 = y_0,
  times = times
)

# Plot the per-capita unregulated birth and loss rates.
plot(times, parms[["mu"]](times), type="l", lwd=2,
     xlab="Time (year)", ylab="Demographic rate (/year)")
abline(h=parms[["b"]], col="red", lwd=2)
legend("topright", col=c("red", "black"), lwd=2, bty="n",
       legend=c("Birth rate", "Loss rate"))

# Plot the population size.
plot(times, state_sens_out[["state"]][,1], type="l", lwd=2,
     xlab="Time (year)", ylab="Population size y")

# Plot the time-dependent state sensitivity. Peaks at around t=10, which is
# roughly when mu and b intersects, so the population has just become
# self-sustaining.
plot(times, state_sens_out[["tdss"]][,1], type="l", lwd=2,
     xlab="Time (year)", ylab="State sensitivity of y")



# ------------------------------------------------------
# Calculating time-dependent parameter sensitivities.
# ------------------------------------------------------
parm_sens_out = parm_sens(
  state_sens_out = state_sens_out
)

# Plot the parameter sensitivity of sigma.
plot(times, parm_sens_out[["tdps"]][["sigma"]][,1], type="l", lwd=2,
     xlab="Time (year)", ylab="Param. sensitivity of sigma")
```

Index

approxfun, *6*

grad, *6*, *7*

jacobian, *2*, *3*, *6*, *7*

ode, *5–8*

parm_sens, 1, *2–5*, *7*, *8*

splinefun, *6*
state_sens, *1–3*, 4, *7*