Package 'ctsmTMB'

August 28, 2025

Type Package

Title Continuous Time Stochastic Modelling using Template Model Builder

Version 1.0.1

Date 2025-08-27

Description Perform state and parameter inference, and forecasting, in stochastic state-space systems using the 'ctsmTMB' class. This class, built with the 'R6' package, provides a user-friendly interface for defining and handling state-space models. Inference is based on maximum likelihood estimation, with derivatives efficiently computed through automatic differentiation enabled by the 'TMB'/'RTMB' packages (Kristensen et al., 2016) <doi:10.18637/jss.v070.i05>. The available inference methods include Kalman filters, in addition to a Laplace approximation-based smoothing method. For further details of these methods refer to the documentation of the 'CTSMR' package <https://ctsm.info/ctsmr-reference.pdf> and Thygesen (2025) <doi:10.48550/arXiv.2503.21358>. Forecasting capabilities include moment predictions and stochastic path simulations, both implemented in 'C++' using 'Rcpp' (Eddelbuettel et al., 2018) <doi:10.1080/00031305.2017.1375990> for computational efficiency.

License GPL-3

URL https://github.com/phillipbvetter/ctsmTMB

BugReports https://github.com/phillipbvetter/ctsmTMB/issues

Depends R (>= 4.0.0)

Imports Deriv, geomtextpath, ggfortify, ggplot2, graphics, grDevices, Matrix, patchwork, R6, RcppXPtrUtils, RTMB (>= 1.7), stats, stringr, TMB, utils

Suggests knitr, rmarkdown, testthat (>= 3.0.0)

LinkingTo Rcpp, RcppEigen, zigg

VignetteBuilder knitr

Config/testthat/edition 3

Encoding UTF-8

LazyData true

RoxygenNote 7.3.2

NeedsCompilation yes

Author Phillip Vetter [aut, cre, cph],
Jan Møller [ctb],
Uffe Thygesen [ctb],
Peder Bacher [ctb],
Henrik Madsen [ctb]

Maintainer Phillip Vetter <pbrve@dtu.dk>

Repository CRAN

Date/Publication 2025-08-27 22:00:02 UTC

Contents

`ctsmTMB` *Methods for the 'ctsmTMB' R6 class*

Description

The following public methods are used to construct a stochastic state space model system, consisting of a set of stochastic differential equations (SDEs), and one or more algebraic observation equations (AOEs). The AOEs are used to infer information about the value of the (latent) states governed by the SDEs, and thus must be functions of at least one state.

Value

The function returns an object of class `R6` and `ctsmTMB`, which can be used to define a stochastic state space system.

Methods

Public methods:

- ctsmTMB$new()
- ctsmTMB$.private()
- ctsmTMB$getPrivateFields()
- ctsmTMB$addSystem()
- ctsmTMB$addObs()
- ctsmTMB$setVariance()
- ctsmTMB$addInput()
- ctsmTMB$setParameter()
- ctsmTMB$setAlgebraics()
- ctsmTMB$setInitialState()
- ctsmTMB$setInitialVarianceScaling()
- ctsmTMB$setLamperti()
- ctsmTMB$setModelname()
- ctsmTMB$setMAP()
- ctsmTMB$setAdvancedSettings()
- ctsmTMB$getSystems()
- ctsmTMB$getObservations()
- ctsmTMB$getVariances()
- ctsmTMB$getAlgebraics()
- ctsmTMB$getInitialState()
- ctsmTMB$getParameters()
- ctsmTMB$getTimers()
- ctsmTMB$getEstimate()
- ctsmTMB$getLikelihood()
- ctsmTMB$getPrediction()
- ctsmTMB$getSimulation()
- ctsmTMB$filter()
- ctsmTMB$smoother()
- ctsmTMB$estimate()
- ctsmTMB$likelihood()
- ctsmTMB$predict()
- ctsmTMB$simulate()
- ctsmTMB$print()
- ctsmTMB$clone()

Method `new()`**:** Initialize private fields

Usage:

```
ctsmTMB$new()
```

Method `.private()`**:** Extract the private fields of a ctsmTMB model object. Primarily used for debugging.

Usage:

```
ctsmTMB$.private()
```

Method `getPrivateFields()`**:** Extract the private fields of a ctsmTMB model object. Primarily used for debugging.

Usage:

```
ctsmTMB$getPrivateFields()
```

Method `addSystem()`**:** Define stochastic differential equation(s) on the form
`d<state> ~ f(t,<states>, <inputs>) * dt + g(t, <states>, <inputs>) * dw`

Usage:

```
ctsmTMB$addSystem(form, ...)
```

Arguments:

`form` a formula specifying a stochastic differential equation

`...` additional formulas similar to `form` for specifying multiple equations at once.

Method `addObs()`**:** Define algebraic observation equations on the form
`<observation> ~ h(t, <states>, <inputs>) + e)`
where `h` is the observation function, and `e` is normally distributed noise with zero mean.
This function only specifies the observation name, and its mean through `h`.

Usage:

```
ctsmTMB$addObs(form, ..., obsnames = NULL)
```

Arguments:

`form` a formula specifying an observation equation

`...` additional formulas similar to `form` for specifying multiple equations at once.

`obsnames` character vector specifying the name of the observation. This is used when the left-hand side of `form` consists of more than just a single variable (of class 'call').

Method `setVariance()`**:** Specify the variance of an observation equation.

A defined observation variable `y` in e.g. `addObs(y ~ h(t,<states>,<inputs>)` is perturbed by Gaussian noise with zero mean and variance to-be specified using `setVariance(y ~ p(t,<states>,<inputs>)`. We can for instance declare `setVariance(y ~ sigma_x^2` where `sigma_x` is a fixed effect parameter to be declared through `setParameter`.

Usage:

```
ctsmTMB$setVariance(form, ...)
```

Arguments:

`form` formula class specifying the observation equation to be added to the system.

`...` additional formulas identical to `form` to specify multiple observation equations at a time.

Method `addInput()`**:** Declare variables as data inputs

Declare whether a variable contained in system, observation or observation variance equations is an input variable. If e.g. the system equation contains an input variable `u` then it is declared using `addInput(u)`. The input `u` must be contained in the data.frame `.data` provided when calling the `estimate` or `predict` methods.

Usage:

```
ctsmTMB$addInput(...)
```

Arguments:

`...` variable names that specifies the name of input variables in the defined system.

Method `setParameter()`**:** Declare which variables that are (fixed effects) parameters in the specified model, and specify the initial optimizer guess, as well as lower / upper bounds during optimization. There are two ways to declare parameters:

1. You can declare parameters using formulas i.e. `setParameter( theta = c(1,0,10), mu = c(0,-10,10) )`. The first value is the initial value for the optimizer, the second value is the lower optimization bound and the third value is the upper optimization bound.
2. You can provide a 3-column matrix where rows corresponds to different parameters, and the parameter names are provided as rownames of the matrix. The columns values corresponds to the description in the vector format above.

Usage:

```
ctsmTMB$setParameter(...)
```

Arguments:

`...` a named vector or matrix as described above.

Method `setAlgebraics()`**:** Add algebraic relations.

Algebraic relations is a convenient way to transform parameters in your equations. In the Ornstein-Uhlenbeck process the rate parameter `theta` is always positive, so estimation in the log-domain is a good idea. Instead of writing `exp(theta)` directly in the system equation one can transform into the log domain using the algebraic relation `setAlgebraics(theta ~ exp(logtheta))`. All instances of `theta` is replaced by `exp(logtheta)` when compiling the C++ function. Note that you must provide values for `logtheta` now instead of `theta` when declaring parameters through `setParameter`

Usage:

```
ctsmTMB$setAlgebraics(form, ...)
```

Arguments:

`form` algebraic formula

`...` additional formulas

Method `setInitialState()`**:** Declare the initial state values i.e. mean and covariance for the system states.

Usage:

```
ctsmTMB$setInitialState(initial.state)
```

Arguments:

`initial.state` a named list of two entries 'x0' and 'p0' containing the initial state and covariance of the state.

Method `setInitialVarianceScaling()`**:** A scalar value that is multiplied onto the estimated initial state covariance matrix. The scaling is only applied when the initial state/cov is estimated, not when it is set by the user.

Usage:

```
ctsmTMB$setInitialVarianceScaling(scaling)
```

Arguments:

`scaling` a numeric scalar value.

Method `setLamperti()`**:** Set a Lamperti Transformation

If the provided system equations have state dependent diffusion in a few available ways then it is advantageous to perform a transformation to remove the state dependence. This comes at the cost of a more complicated drift function. The following types of state-dependence is currently supported

1. 'identity' - when the diffusion is state-independent (default)
2. 'log' - when the diffusion is proportional to to x * dw
3. 'logit' - when the diffusion is proportional to x * (1-x) * dw
4. 'sqrt-logit' - when the diffusion is proportional to sqrt(x * (1-x)) * dw

Usage:

```
ctsmTMB$setLamperti(transforms, states = NULL)
```

Arguments:

`transforms` character vector - one of either "identity, "log", "logit", "sqrt-logit"

`states` a vector of the state names for which the specified transformations should be applied to.

Method `setModelname()`**:** Set modelname used to create the C++ file for TMB

When calling `TMB::MakeADFun` the (negative log) likelihood function is created in the directory specified by the `setCppfilesDirectory` method with name `<modelname>.cpp`

Usage:

```
ctsmTMB$setModelname(name)
```

Arguments:

`name` string defining the model name.

Method `setMAP()`**:** Enable maximum a posterior (MAP) estimation.

Adds a maximum a posterior contribution to the (negative log) likelihood function by evaluating the fixed effects parameters in a multivariate Gaussian with `mean` and `covariance` as provided.

Usage:

```
ctsmTMB$setMAP(mean, cov)
```

Arguments:

`mean` mean vector of the Gaussian prior parameter distribution

`cov` covariance matrix of the Gaussian prior parameter distribution

Method `setAdvancedSettings()`**:** Enable maximum a posterior (MAP) estimation.

Adds a maximum a posterior contribution to the (negative log) likelihood function by evaluating the fixed effects parameters in a multivariate Gaussian with `mean` and `covariance` as provided.

Usage:

```
ctsmTMB$setAdvancedSettings(forceAD = TRUE)
```

Arguments:

`forceAD` a boolean indicating whether to use state space functions that take advantage of the RTMB::AD(...,force=TRUE) hack which reduces compilation time call to MakeADFun by 20%. This breaks some functionalities such as REPORT.

Method `getSystems()`**:** Retrieve system equations.

Usage:

```
ctsmTMB$getSystems()
```

Method `getObservations()`**:** Retrieve observation equations.

Usage:

```
ctsmTMB$getObservations()
```

Method `getVariances()`**:** Retrieve observation variances

Usage:

```
ctsmTMB$getVariances()
```

Method `getAlgebraics()`**:** Retrieve algebraic relations

Usage:

```
ctsmTMB$getAlgebraics()
```

Method `getInitialState()`**:** Retrieve initially set state and covariance

Usage:

```
ctsmTMB$getInitialState()
```

Method `getParameters()`**:** Get initial (and estimated) parameters.

Usage:

```
ctsmTMB$getParameters(type = "all", value = "all")
```

Arguments:

`type` one of "all", free" or "fixed" parameters.

`value` one of "all", initial", "estimate", "lower" or "upper"

Method `getTimers()`**:** Retrieve initially timers

Usage:

```
ctsmTMB$getTimers()
```

Method `getEstimate()`**:** Retrieve initially set state and covariance

Usage:

```
ctsmTMB$getEstimate()
```

Method `getLikelihood()`**:** Retrieve initially set state and covariance

Usage:

```
ctsmTMB$getLikelihood()
```

Method `getPrediction()`**:** Retrieve initially set state and covariance

Usage:

```
ctsmTMB$getPrediction()
```

Method `getSimulation()`**:** Retrieve initially set state and covariance

Usage:

```
ctsmTMB$getSimulation()
```

Method `filter()`**:** Perform state filtering (or smoothing for the 'laplace' method)

Usage:

```
ctsmTMB$filter(
  data,
  pars = NULL,
  method = "ekf",
  ode.solver = "euler",
  ode.timestep = diff(data$t),
  loss = "quadratic",
  loss_c = NULL,
  ukf.hyperpars = c(1, 0, 3),
  initial.state = self$getInitialState(),
  laplace.residuals = FALSE,
  estimate.initial.state = FALSE,
  use.cpp = FALSE,
  silent = FALSE,
  ...
)
```

Arguments:

`data` data.frame containing time-vector 't', observations and inputs. The observations can take `NA`-values.

`pars` fixed parameter vector parsed to the objective function for prediction/filtration. The default parameter values used are the initial parameters provided through `setParameter`, unless the `estimate`

`method` character vector specifying the filtering method used for state/likelihood calculations. Must be one of either "lkf", "ekf", "laplace".

`ode.solver` Sets the ODE solver used in the Kalman Filter methods for solving the moment differential equations. The default "euler" is the Forward Euler method, alternatively the classical 4th order Runge Kutta method is available via "rk4".

`ode.timestep` numeric value. Sets the time step-size in numerical filtering schemes. The defined step-size is used to calculate the number of steps between observation time-points as defined by the provided `data`. If the calculated number of steps is larger than N.01 where N is an integer, then the time-step is reduced such that exactly N+1 steps is taken between observations The step-size is used in the two following ways depending on the chosen method:

1. Kalman filters: The time-step is used as the step-size in the numerical Forward-Euler scheme to compute the prior state mean and covariance estimate as the final time solution to the first and second order moment differential equations.

2. TMB method: The time-step is used as the step-size in the Euler-Maruyama scheme for simulating a sample path of the stochastic differential equation, which serves to link together the latent (random effects) states.

`loss` character vector. Sets the loss function type (only implemented for the kalman filter methods). The loss function is per default quadratic in the one-step residuals as is natural when the Gaussian (negative log) likelihood is evaluated, but if the tails of the distribution is considered too small i.e. outliers are weighted too much, then one can choose loss functions that accounts for this. The three available types available:

1. Quadratic loss (`quadratic`).
2. Quadratic-Linear (`huber`)
3. Quadratic-Constant (`tukey`)

The cutoff for the Huber and Tukey loss functions are determined from a provided cutoff parameter `loss_c`. The implementations of these losses are approximations (pseudo-huber and sigmoid approximation respectively) for smooth derivatives.

`loss_c` cutoff value for huber and tukey loss functions. Defaults to `c=3`

`ukf.hyperpars` The hyperparameters alpha, beta, and kappa used for sigma points and weights construction in the Unscented Kalman Filter.

`initial.state` a named list of two entries 'x0' and 'p0' containing the initial state and covariance of the state

`laplace.residuals` boolean - whether or not to calculate one-step ahead residuals using the method of oneStepPredict.

`estimate.initial.state` boolean value. When TRUE the initial state and covariance matrices are estimated as the stationary solution of the linearized mean and covariance differential equations. When the system contains time-varying inputs, the first element of these is used.

`use.cpp` a boolean to indicate whether to use C++ to perform calculations

`silent` logical value whether or not to suppress printed messages such as 'Checking Data', 'Building Model', etc. Default behaviour (FALSE) is to print the messages.

`...` additional arguments

Method `smoother()`**:** Perform state filtering (or smoothing for the 'laplace' method)

Usage:

```
ctsmTMB$smoother(
  data,
  pars = NULL,
  method = "ekf",
  ode.solver = "euler",
  ode.timestep = diff(data$t),
  loss = "quadratic",
  loss_c = NULL,
  initial.state = self$getInitialState(),
  laplace.residuals = FALSE,
  estimate.initial.state = FALSE,
  silent = FALSE,
  ...
)
```

Arguments:

`data` data.frame containing time-vector 't', observations and inputs. The observations can take `NA`-values.

`pars` fixed parameter vector parsed to the objective function for prediction/filtration. The default parameter values used are the initial parameters provided through `setParameter`, unless the `estimate`

`method` character vector specifying the filtering method used for state/likelihood calculations. Must be one of either "lkf", "ekf", "laplace".

`ode.solver` Sets the ODE solver used in the Kalman Filter methods for solving the moment differential equations. The default "euler" is the Forward Euler method, alternatively the classical 4th order Runge Kutta method is available via "rk4".

`ode.timestep` numeric value. Sets the time step-size in numerical filtering schemes. The defined step-size is used to calculate the number of steps between observation time-points as defined by the provided `data`. If the calculated number of steps is larger than N.01 where N is an integer, then the time-step is reduced such that exactly N+1 steps is taken between observations The step-size is used in the two following ways depending on the chosen method:

1. Kalman filters: The time-step is used as the step-size in the numerical Forward-Euler scheme to compute the prior state mean and covariance estimate as the final time solution to the first and second order moment differential equations.
2. TMB method: The time-step is used as the step-size in the Euler-Maruyama scheme for simulating a sample path of the stochastic differential equation, which serves to link together the latent (random effects) states.

`loss` character vector. Sets the loss function type (only implemented for the kalman filter methods). The loss function is per default quadratic in the one-step residuals as is natural when the Gaussian (negative log) likelihood is evaluated, but if the tails of the distribution is considered too small i.e. outliers are weighted too much, then one can choose loss functions that accounts for this. The three available types available:

1. Quadratic loss (`quadratic`).
2. Quadratic-Linear (`huber`)
3. Quadratic-Constant (`tukey`)

The cutoff for the Huber and Tukey loss functions are determined from a provided cutoff parameter `loss_c`. The implementations of these losses are approximations (pseudo-huber and sigmoid approximation respectively) for smooth derivatives.

`loss_c` cutoff value for huber and tukey loss functions. Defaults to `c=3`

`initial.state` a named list of two entries 'x0' and 'p0' containing the initial state and covariance of the state

`laplace.residuals` boolean - whether or not to calculate one-step ahead residuals using the method of oneStepPredict.

`estimate.initial.state` boolean value. When TRUE the initial state and covariance matrices are estimated as the stationary solution of the linearized mean and covariance differential equations. When the system contains time-varying inputs, the first element of these is used.

`silent` logical value whether or not to suppress printed messages such as 'Checking Data', 'Building Model', etc. Default behaviour (FALSE) is to print the messages.

`...` additional arguments

Method `estimate()`**:** Estimate the fixed effects parameters in the specified model.

Usage:

```
ctsmTMB$estimate(
  data,
  method = "ekf",
  ode.solver = "euler",
  ode.timestep = diff(data$t),
  loss = "quadratic",
  loss_c = NULL,
  ukf.hyperpars = c(1, 0, 3),
  initial.state = self$getInitialState(),
  trace = 10,
  control = list(trace = trace, iter.max = 1e+05, eval.max = 1e+05),
  use.hessian = FALSE,
  laplace.residuals = FALSE,
  unconstrained.optim = FALSE,
  estimate.initial.state = FALSE,
  silent = FALSE,
  compile = FALSE,
  ...
)
```

Arguments:

`data` data.frame containing time-vector 't', observations and inputs. The observations can take `NA`-values.

`method` character vector specifying the filtering method used for state/likelihood calculations. Must be one of either "lkf", "ekf", "laplace".

`ode.solver` Sets the ODE solver used in the Kalman Filter methods for solving the moment differential equations. The default "euler" is the Forward Euler method, alternatively the classical 4th order Runge Kutta method is available via "rk4".

`ode.timestep` numeric value. Sets the time step-size in numerical filtering schemes. The defined step-size is used to calculate the number of steps between observation time-points as defined by the provided `data`. If the calculated number of steps is larger than N.01 where N is an integer, then the time-step is reduced such that exactly N+1 steps is taken between observations The step-size is used in the two following ways depending on the chosen method:

1. Kalman filters: The time-step is used as the step-size in the numerical Forward-Euler scheme to compute the prior state mean and covariance estimate as the final time solution to the first and second order moment differential equations.
2. TMB method: The time-step is used as the step-size in the Euler-Maruyama scheme for simulating a sample path of the stochastic differential equation, which serves to link together the latent (random effects) states.

`loss` character vector. Sets the loss function type (only implemented for the kalman filter methods). The loss function is per default quadratic in the one-step residuals as is natural when the Gaussian (negative log) likelihood is evaluated, but if the tails of the distribution is considered too small i.e. outliers are weighted too much, then one can choose loss functions that accounts for this. The three available types available:

1. Quadratic loss (`quadratic`).
2. Quadratic-Linear (`huber`)

3. Quadratic-Constant (tukey)

The cutoff for the Huber and Tukey loss functions are determined from a provided cutoff parameter loss_c. The implementations of these losses are approximations (pseudo-huber and sigmoid approximation respectively) for smooth derivatives.

loss_c cutoff value for huber and tukey loss functions. Defaults to c=3

ukf.hyperpars The hyperparameters alpha, beta, and kappa used for sigma points and weights construction in the Unscented Kalman Filter.

initial.state a named list of two entries 'x0' and 'p0' containing the initial state and covariance of the state

trace integer passed to control which determines number of steps between each print-out during optimization (use 0 to disable tracing print-outs).

control list of control parameters parsed to nlminb as its control argument. See ?stats::nlminb for more information

use.hessian boolean value. The default (TRUE) causes the optimization algorithm stats::nlminb to use the fixed effects hessian of the (negative log) likelihood when performing the optimization. This feature is only available for the kalman filter methods without any random effects.

laplace.residuals boolean - whether or not to calculate one-step ahead residuals using the method of oneStepPredict.

unconstrained.optim boolean value. When TRUE then the optimization is carried out unconstrained i.e. without any of the parameter bounds specified during setParameter.

estimate.initial.state boolean value. When TRUE the initial state and covariance matrices are estimated as the stationary solution of the linearized mean and covariance differential equations. When the system contains time-varying inputs, the first element of these is used.

silent logical value whether or not to suppress printed messages such as 'Checking Data', 'Building Model', etc. Default behaviour (FALSE) is to print the messages.

compile boolean for (re)compiling the objective C++ file, used for methods ending with _cpp.

... additional arguments

Method likelihood()**:** Construct and extract function handlers for the negative log likelihood function.

The handlers from TMB's MakeADFun are constructed and returned. This enables the user to e.g. choose their own optimization algorithm, or just have more control of the optimization workflow.

Usage:

```
ctsmTMB$likelihood(
  data,
  method = "ekf",
  ode.solver = "euler",
  ode.timestep = diff(data$t),
  loss = "quadratic",
  loss_c = NULL,
  ukf.hyperpars = c(1, 0, 3),
  initial.state = self$getInitialState(),
  estimate.initial.state = FALSE,
  silent = FALSE,
  compile = FALSE,
```

```
  ...
)
```

Arguments:

`data` a data.frame containing time-vector 't', observations and inputs. The observations can take `NA`-values.

`method` character vector specifying the filtering method used for state/likelihood calculations. Must be one of either "lkf", "ekf", "laplace".

`ode.solver` Sets the ODE solver used in the Kalman Filter methods for solving the moment differential equations. The default "euler" is the Forward Euler method, alternatively the classical 4th order Runge Kutta method is available via "rk4".

`ode.timestep` the time-step used in the filtering schemes. The time-step has two different uses depending on the chosen method.

1. Kalman Filters: The time-step is used when numerically solving the moment differential equations.
2. Laplace Approximation: The time-step is used in the Euler-Maruyama simulation scheme for simulating a sample path of the stochastic differential equation, which serves to link together the latent (random effects) states.

The defined step-size is used to calculate the number of steps between observation time-points as defined by the provided `data`. If the calculated number of steps is larger than N.01 where N is an integer, then the time-step is reduced such that exactly N+1 steps is taken between observations The step-size is used in the two following ways depending on the chosen method:

1. Kalman filters: The time-step is used as the step-size in the numerical Forward-Euler scheme to compute the prior state mean and covariance estimate as the final time solution to the first and second order moment differential equations.
2. TMB method: The time-step is used as the step-size in the Euler-Maruyama scheme for simulating a sample path of the stochastic differential equation, which serves to link together the latent (random effects) states.

`loss` character vector. Sets the loss function type (only implemented for the kalman filter methods). The loss function is per default quadratic in the one-step residuals as is natural when the Gaussian (negative log) likelihood is evaluated, but if the tails of the distribution is considered too small i.e. outliers are weighted too much, then one can choose loss functions that accounts for this. The three available types available:

1. Quadratic loss (`quadratic`).
2. Quadratic-Linear (`huber`)
3. Quadratic-Constant (`tukey`)

The cutoff for the Huber and Tukey loss functions are determined from a provided cutoff parameter `loss_c`. The implementations of these losses are approximations (pseudo-huber and sigmoid approximation respectively) for smooth derivatives.

`loss_c` cutoff value for huber and tukey loss functions. Defaults to `c=3`

`ukf.hyperpars` The hyperparameters alpha, beta, and kappa used for sigma points and weights construction in the Unscented Kalman Filter.

`initial.state` a named list of two entries 'x0' and 'p0' containing the initial state and covariance of the state

`estimate.initial.state` boolean value. When TRUE the initial state and covariance matrices are estimated as the stationary solution of the linearized mean and covariance differential equations. When the system contains time-varying inputs, the first element of these is used.

`silent` logical value whether or not to suppress printed messages such as 'Checking Data', 'Building Model', etc. Default behaviour (FALSE) is to print the messages.

`compile` boolean for (re)compiling the objective C++ file, used for methods ending with `_cpp`.

`...` additional arguments

Method `predict()`**:** Perform prediction/filtration to obtain state mean and covariance estimates. The predictions are obtained by solving the moment equations `n.ahead` steps forward in time when using the current step posterior state estimate as the initial condition.

Usage:

```
ctsmTMB$predict(
  data,
  pars = NULL,
  method = "ekf",
  ode.solver = "euler",
  ode.timestep = diff(data$t),
  k.ahead = nrow(data) - 1,
  return.k.ahead = 0:k.ahead,
  return.covariance = TRUE,
  initial.state = self$getInitialState(),
  estimate.initial.state = private$estimate.initial,
  use.cpp = FALSE,
  silent = FALSE,
  ...
)
```

Arguments:

`data` data.frame containing time-vector 't', observations and inputs. The observations can take `NA`-values.

`pars` fixed parameter vector parsed to the objective function for prediction/filtration. The default parameter values used are the initial parameters provided through `setParameter`, unless the `estimate` function has been run, then the default values will be those at the found optimum.

`method` The prediction method

`ode.solver` Sets the ODE solver used in the Kalman Filter methods for solving the moment differential equations. The default "euler" is the Forward Euler method, alternatively the classical 4th order Runge Kutta method is available via "rk4".

`ode.timestep` numeric value. Sets the time step-size in numerical filtering schemes. The defined step-size is used to calculate the number of steps between observation time-points as defined by the provided `data`. If the calculated number of steps is larger than N.01 where N is an integer, then the time-step is reduced such that exactly N+1 steps is taken between observations The step-size is used in the two following ways depending on the chosen method:

1. Kalman filters: The time-step is used as the step-size in the numerical Forward-Euler scheme to compute the prior state mean and covariance estimate as the final time solution to the first and second order moment differential equations.
2. TMB method: The time-step is used as the step-size in the Euler-Maruyama scheme for simulating a sample path of the stochastic differential equation, which serves to link together the latent (random effects) states.

`k.ahead` integer specifying the desired number of time-steps (as determined by the provided data time-vector) for which predictions are made (integrating the moment ODEs forward in time without data updates).

`return.k.ahead` numeric vector of integers specifying which n.ahead predictions to that should be returned.

`return.covariance` boolean value to indicate whether the covariance (instead of the correlation) should be returned.

`initial.state` a named list of two entries 'x0' and 'p0' containing the initial state and covariance of the state

`estimate.initial.state` bool - stationary estimation of initial mean and covariance

`use.cpp` a boolean to indicate whether to use C++ to perform calculations

`silent` logical value whether or not to suppress printed messages such as 'Checking Data', 'Building Model', etc. Default behaviour (FALSE) is to print the messages.

`...` additional arguments

Returns: A data.frame that contains for each time step the posterior state estimate at that time.step (`k = 0`), and the prior state predictions (`k = 1,...,n.ahead`). If `return.covariance = TRUE` then the state covariance/correlation matrix is returned, otherwise only the marginal variances are returned.

Method `simulate()`**:** Perform prediction/filtration to obtain state mean and covariance estimates. The predictions are obtained by solving the moment equations `n.ahead` steps forward in time when using the current step posterior state estimate as the initial condition.

Usage:

```
ctsmTMB$simulate(
  data,
  pars = NULL,
  use.cpp = FALSE,
  cpp.seed = NULL,
  method = "ekf",
  ode.solver = "rk4",
  ode.timestep = diff(data$t),
  simulation.timestep = diff(data$t),
  k.ahead = nrow(data) - 1,
  return.k.ahead = 0:k.ahead,
  n.sims = 100,
  initial.state = self$getInitialState(),
  estimate.initial.state = private$estimate.initial,
  silent = FALSE,
  ...
)
```

Arguments:

`data` data.frame containing time-vector 't', observations and inputs. The observations can take `NA`-values.

`pars` fixed parameter vector parsed to the objective function for prediction/filtration. The default parameter values used are the initial parameters provided through `setParameter`, unless the `estimate` function has been run, then the default values will be those at the found optimum.

`use.cpp` a boolean to indicate whether to use C++ to perform calculations

`cpp.seed` an integer seed value to control RNG normal draws on the C++ side.

`method` 1. The natural TMB-style formulation where latent states are considered random effects and are integrated out using the Laplace approximation. This method only yields the gradient of the (negative log) likelihood function with respect to the fixed effects for optimization. The method is slower although probably has some precision advantages, and allows for non-Gaussian observation noise (not yet implemented). One-step / K-step residuals are not yet available in the package.

2. (Continuous-Discrete) Extended Kalman Filter where the system dynamics are linearized to handle potential non-linearities. This is computationally the fastest method.

3. (Continuous-Discrete) Unscented Kalman Filter. This is a higher order non-linear Kalman Filter which improves the mean and covariance estimates when the system display high nonlinearity, and circumvents the necessity to compute the Jacobian of the drift and observation functions.

All package features are currently available for the kalman filters, while TMB is limited to parameter estimation. In particular, it is straight-forward to obtain k-step-ahead predictions with these methods (use the `predict` S3 method), and stochastic simulation is also available in the cases where long prediction horizons are sought, where the normality assumption will be inaccurate

`ode.solver` Sets the ODE solver used in the Kalman Filter methods for solving the moment differential equations. The default "euler" is the Forward Euler method, alternatively the classical 4th order Runge Kutta method is available via "rk4".

`ode.timestep` numeric value. Sets the time step-size in numerical filtering schemes. The defined step-size is used to calculate the number of steps between observation time-points as defined by the provided `data`. If the calculated number of steps is larger than N.01 where N is an integer, then the time-step is reduced such that exactly N+1 steps is taken between observations The step-size is used in the two following ways depending on the chosen method:

1. Kalman filters: The time-step is used as the step-size in the numerical Forward-Euler scheme to compute the prior state mean and covariance estimate as the final time solution to the first and second order moment differential equations.

2. TMB method: The time-step is used as the step-size in the Euler-Maruyama scheme for simulating a sample path of the stochastic differential equation, which serves to link together the latent (random effects) states.

`simulation.timestep` timestep used in the euler-maruyama scheme

`k.ahead` integer specifying the desired number of time-steps (as determined by the provided data time-vector) for which predictions are made (integrating the moment ODEs forward in time without data updates).

`return.k.ahead` numeric vector of integers specifying which n.ahead predictions to that should be returned.

`n.sims` number of simulations

`initial.state` a named list of two entries 'x0' and 'p0' containing the initial state and covariance of the state

`estimate.initial.state` bool - stationary estimation of initial mean and covariance

`silent` logical value whether or not to suppress printed messages such as 'Checking Data', 'Building Model', etc. Default behaviour (FALSE) is to print the messages.

`...` additional arguments

`return.covariance` boolean value to indicate whether the covariance (instead of the correlation) should be returned.

Returns: A data.frame that contains for each time step the posterior state estimate at that time.step (`k = 0`), and the prior state predictions (`k = 1,...,n.ahead`). If `return.covariance = TRUE` then the state covariance/correlation matrix is returned, otherwise only the marginal variances are returned.

Method `print()`**:** Function to print the model object

Usage:

```
ctsmTMB$print()
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
ctsmTMB$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
library(ctsmTMB)
model <- ctsmTMB$new()

# adding a single system equations
model$addSystem(dx ~ theta * (mu+u-x) * dt + sigma_x*dw)

# adding an observation equation and setting variance
model$addObs(y ~ x)
model$setVariance(y ~ sigma_y^2)

# add model input
model$addInput(u)

# add parameters
model$setParameter(
  theta   = c(initial = 1, lower=1e-5, upper=50),
  mu      = c(initial=1.5, lower=0, upper=5),
  sigma_x = c(initial=1, lower=1e-10, upper=30),
  sigma_y = 1e-2
)

# set the model initial state
model$setInitialState(list(1,1e-1))

# extract the likelihood handlers
nll <- model$likelihood(data=Ornstein)

# calculate likelihood, gradient and hessian w.r.t parameters
nll$fn(nll$par)
nll$gr(nll$par)
```

```
nll$he(nll$par)

# estimate the parameters using an extended kalman filter
fit <- model$estimate(data=Ornstein)

# perform moment predictions
pred <- model$predict(data=Ornstein)

# perform stochatic simulations
sim <- model$simulate(data=Ornstein, n.sims=10)
```

`Ornstein` *Sample from a simulated Ornstein-Uhlenbeck process with time-dependent mean*

Description

The data was simulated using a standard Euler-Maruyama method.

The simulated process is governed by the SDE #' dx ~ theta * (mu + u - x) * dt + sigma_x * dw

The parameters used for simulation were theta = 2, mu = 0.5, sigma_x = 1.358, sigma_y = 1e-8

The simulation time-step was 1e-3, and observation time-step 1e-1. The simulation was taken from t = 0..20

Usage

```
Ornstein
```

Format

A data frame of 201 rows and 3 columns. The columns represent the variables: `t` (time), `y` (observation) and `u` (input).

`plot.ctsmTMB.fit` *This function creates residual plots for an estimated ctsmTMB object*

Description

This function creates residual plots for an estimated ctsmTMB object

Usage

```
## S3 method for class 'ctsmTMB.fit'
plot(
  x,
  print.plot = 1,
  type = "residuals",
  state.type = "prior",
  against.obs = NULL,
  ggtheme = getggplot2theme(),
  ylims = c(NA, NA),
  residual.burnin = 0L,
  residual.vs.obs.and.inputs = FALSE,
  ...
)
```

Arguments

x	A R6 ctsmTMB fit object
print.plot	a single integer determining which element out of all states/observations (depending on the argument to type).
type	a character vector either 'residuals' or 'states' determining what to plot.
state.type	a character vector either 'prior', 'posterior' or 'smoothed' determining what kind of states to plot.
against.obs	name of an observation to plot state predictions against.
ggtheme	ggplot2 theme to use for creating the ggplot.
ylims	limits on the y-axis for residual time-series plot
residual.burnin	integer N to remove the first N residuals
residual.vs.obs.and.inputs	the residual plots also include a new window with time-series plots of residuals, associated observations and inputs
...	additional arguments

Value

a (list of) ggplot residual plot(s)

Examples

```
library(ctsmTMB)
model <- ctsmTMB$new()

# create model
model$addSystem(dx ~ theta * (mu+u-x) * dt + sigma_x*dw)
model$addObs(y ~ x)
model$setVariance(y ~ sigma_y^2)
model$addInput(u)
```

```
model$setParameter(
  theta   = c(initial = 1, lower=1e-5, upper=50),
  mu      = c(initial=1.5, lower=0, upper=5),
  sigma_x = c(initial=1, lower=1e-10, upper=30),
  sigma_y = 1e-2
)
model$setInitialState(list(1,1e-1))

# fit model to data
fit <- model$estimate(Ornstein)

# plot residuals
## Not run: plot(fit)

# plot filtered states
## Not run: plot(fit, type="states")
```

`plot.ctsmTMB.pred` *Plot of k-step predictions from a ctsmTMB prediction object*

Description

Plot of k-step predictions from a ctsmTMB prediction object

Usage

```
## S3 method for class 'ctsmTMB.pred'
plot(
  x,
  y,
  k.ahead = unique(x[["states"]][["k.ahead"]]),
  state.name = NULL,
  type = "states",
  against = NULL,
  ...
)
```

Arguments

`x`	a ctsmTMB.pred object
`y`	not used
`k.ahead`	an integer indicating which k-ahead predictions to plot
`state.name`	a string indicating which states to plot
`type`	one of 'states' or 'observations', to plot
`against`	name of an observations to plot predictions against
`...`	additional arguments

Value

A plot of predicted states

Examples

```
library(ctsmTMB)
model <- ctsmTMB$new()

# create model
model$addSystem(dx ~ theta * (mu+u-x) * dt + sigma_x*dw)
model$addObs(y ~ x)
model$setVariance(y ~ sigma_y^2)
model$addInput(u)
model$setParameter(
  theta   = c(initial = 1, lower=1e-5, upper=50),
  mu      = c(initial=1.5, lower=0, upper=5),
  sigma_x = c(initial=1, lower=1e-10, upper=30),
  sigma_y = 1e-2
)
model$setInitialState(list(1,1e-1))

# fit model to data
fit <- model$estimate(Ornstein)

# perform moment predictions
pred <- model$predict(Ornstein)

# plot the k.ahead=10 predictions
plot(pred, against="y.data")


# plot filtered states
plot(fit, type="states", against="y")
```

---

plot.ctsmTMB.profile *#' Plot a profile likelihood ctsmTMB object*

Description

#' Plot a profile likelihood ctsmTMB object

Usage

```
## S3 method for class 'ctsmTMB.profile'
plot(x, y, include.opt = TRUE, ...)
```

Arguments

x	a profile.ctsmTMB object
y	not in use
include.opt	boolean which indicates whether or not to include the total likelihood optimizer in the plot.
...	additional arguments

Examples

```
library(ctsmTMB)
model <- ctsmTMB$new()

# create model
model$addSystem(dx ~ theta * (mu+u-x) * dt + sigma_x*dw)
model$addObs(y ~ x)
model$setVariance(y ~ sigma_y^2)
model$addInput(u)
model$setParameter(
  theta   = c(initial = 1, lower=1e-5, upper=50),
  mu      = c(initial=1.5, lower=0, upper=5),
  sigma_x = c(initial=1, lower=1e-10, upper=30),
  sigma_y = 1e-2
)
model$setInitialState(list(1,1e-1))

# fit model to data
fit <- model$estimate(Ornstein)

# calculate profile likelihood
out <- profile(fit,parlist=list(theta=NULL))

# plot profile
plot(out)
```

---

print.ctsmTMB *Basic print of ctsmTMB objects*

Description

Basic print of ctsmTMB objects

Usage

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

Arguments

x	an object of class 'ctsmTMB'
...	additional arguments (not in use)

Value

Print of ctsmTMB model object

Examples

```
library(ctsmTMB)
model <- ctsmTMB$new()

# print empty model
print(model)

# add elements to model and see new print
model$addSystem(dx ~ theta * (mu+u-x) * dt + sigma_x*dw)
model$addObs(y ~ x)
model$setVariance(y ~ sigma_y^2)
model$addInput(u)
model$setParameter(
  theta   = c(initial = 1, lower=1e-5, upper=50),
  mu      = c(initial=1.5, lower=0, upper=5),
  sigma_x = c(initial=1, lower=1e-10, upper=30),
  sigma_y = 1e-2
)
print(model)
```

---

print.ctsmTMB.fit *Basic print of objects ctsmTMB fit objects*

Description

Basic print of objects ctsmTMB fit objects

Usage

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

Arguments

x	a ctsmTMB fit object
...	additional arguments

Value

Print of ctsmTMB fit object

Examples

```
library(ctsmTMB)
model <- ctsmTMB$new()

# create model
model$addSystem(dx ~ theta * (mu+u-x) * dt + sigma_x*dw)
model$addObs(y ~ x)
model$setVariance(y ~ sigma_y^2)
model$addInput(u)
model$setParameter(
  theta   = c(initial = 1, lower=1e-5, upper=50),
  mu      = c(initial=1.5, lower=0, upper=5),
  sigma_x = c(initial=1, lower=1e-10, upper=30),
  sigma_y = 1e-2
)
model$setInitialState(list(1,1e-1))

# fit model to data
fit <- model$estimate(Ornstein)

# print fit
print(fit)
```

`profile.ctsmTMB.fit` *#' Performs full multi-dimensional profile likelihood calculations*

Description

#' Performs full multi-dimensional profile likelihood calculations

Usage

```
## S3 method for class 'ctsmTMB.fit'
profile(
  fitted,
  parlist,
  grid.size = rep(10, length(parlist)),
  grid.qnt = rep(3, length(parlist)),
  hessian = FALSE,
  silent = FALSE,
  control = list(trace = 0, iter.max = 1000, eval.max = 1000),
  ...
)
```

Arguments

`fitted`	a ctmsTMB fit object

parlist	a named-list of parameters to profile over. The user can either supply grid-values in the list or leave it empty. If the any one list is empty then grid-values will be calculated using the estimated parameter mean value and standard deviation.
grid.size	a vector of length(parlist) indicating the number of grid-points along each parameter direction. This is only used if the parlist is empty.
grid.qnt	a vector of length(parlist) determining the width of the grid points from the mean value in multiples of the standard deviation.
hessian	a boolean indicating whether to use the hessian or not during the profile optimization.
silent	boolean whether or not to mute current iteration number the control argument.
control	a list of optimization output controls (see nlminb)
...	various arguments (not in use)

Note

The implementation was modified from that of https://github.com/kaskr/adcomp/blob/master/TMB/R/tmbprofile.R

Examples

```
library(ctsmTMB)
model <- ctsmTMB$new()

# create model
model$addSystem(dx ~ theta * (mu+u-x) * dt + sigma_x*dw)
model$addObs(y ~ x)
model$setVariance(y ~ sigma_y^2)
model$addInput(u)
model$setParameter(
  theta   = c(initial = 1, lower=1e-5, upper=50),
  mu      = c(initial=1.5, lower=0, upper=5),
  sigma_x = c(initial=1, lower=1e-10, upper=30),
  sigma_y = 1e-2
)
model$setInitialState(list(1,1e-1))

# fit model to data
fit <- model$estimate(Ornstein)

# calculate profile likelihood
out <- profile(fit,parlist=list(theta=NULL))
```

---

summary.ctsmTMB.fit *Basic summary of ctsmTMB fit object*

Description

Basic summary of ctsmTMB fit object

Usage

```
## S3 method for class 'ctsmTMB.fit'
summary(object, correlation = FALSE, ...)
```

Arguments

object	a ctsmTMB fit object
correlation	boolean indicating whether or not to display the parameter correlation structure
...	additional arguments

Value

a summary of the estimated ctsmTMB model fit

Examples

```
library(ctsmTMB)
model <- ctsmTMB$new()

# create model
model$addSystem(dx ~ theta * (mu+u-x) * dt + sigma_x*dw)
model$addObs(y ~ x)
model$setVariance(y ~ sigma_y^2)
model$addInput(u)
model$setParameter(
  theta   = c(initial = 1, lower=1e-5, upper=50),
  mu      = c(initial=1.5, lower=0, upper=5),
  sigma_x = c(initial=1, lower=1e-10, upper=30),
  sigma_y = 1e-2
)
model$setInitialState(list(1,1e-1))

# fit model to data
fit <- model$estimate(Ornstein)

# print model summary
summary(fit, correlation=TRUE)
```

Index