Package: nimbleSMC (via r-universe)

September 13, 2024

Title Sequential Monte Carlo Methods for 'nimble'

Description Includes five particle filtering algorithms for use with state space models in the 'nimble' system: 'Auxiliary', 'Bootstrap', 'Ensemble Kalman filter', 'Iterated Filtering 2', and 'Liu-West', as described in Michaud et al. (2021), <doi:10.18637/jss.v100.i03>. A full User Manual is available at <https://r-nimble.org>.

Version 0.11.1

Date 2024-06-14

Maintainer Christopher Paciorek <paciorek@stat.berkeley.edu>

License BSD_3_clause + file LICENSE | GPL (>= 2)

Copyright See COPYRIGHTS file.

Depends R (>= 3.1.2), nimble (>= 1.0.0)

Imports methods

Suggests testthat

URL https://r-nimble.org, https://github.com/nimble-dev/nimbleSMC

Encoding UTF-8

Collate AuxiliaryFilter.R BootstrapFilter.R EnKFilter.R IF2Filter.R LiuWestFilter.R MCMCSamplers.R resamplers.R utils.R zzz.R

RoxygenNote 7.2.3

NeedsCompilation no

Author Nick Michaud [aut], Perry de Valpine [aut], Christopher Paciorek [aut, cre], Daniel Turek [aut], Benjamin R. Goldstein [ctb] (packaging contributions), Dao Nguyen [ctb] (contributions to the IF2 code), The Regents of the University of California [cph]

Date/Publication 2024-06-14 15:40:02 UTC

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

RemoteUrl https://github.com/cran/nimbleSMC

RemoteRef HEAD

RemoteSha 91f2513e7064d1ce397287b1cf66db3e96409cc0

Contents

`buildAuxiliaryFilter` *Create an auxiliary particle filter algorithm to estimate log-likelihood.*

Description

Create an auxiliary particle filter algorithm for a given NIMBLE state space model.

Usage

```
buildAuxiliaryFilter(model, nodes, control = list())
```

Arguments

`model`	A NIMBLE model object, typically representing a state space model or a hidden Markov model.
`nodes`	A character vector specifying the stochastic latent model nodes over which the particle filter will stochastically integrate to estimate the log-likelihood function. All provided nodes must be stochastic. Can be one of three forms: a variable name, in which case all elements in the variable are taken to be latent (e.g., 'x'); an indexed variable, in which case all indexed elements are taken to be latent (e.g., 'x[1:100]' or 'x[1:100, 1:2]'); or a vector of multiple nodes, one per time point, in increasing time order (e.g., c("x[1:2, 1]", "x[1:2, 2]", "x[1:2, 3]", "x[1:2, 4]")).
`control`	A list specifying different control options for the particle filter. Options are described in the details section below.

Details

Each of the `control()` list options are described in detail here:

lookahead The lookahead function used to calculate auxiliary weights. Can choose between `'mean'` and `'simulate'`. Defaults to `'simulate'`.

resamplingMethod The type of resampling algorithm to be used within the particle filter. Can choose between `'default'` (which uses NIMBLE's `rankSample()` function), `'systematic'`, `'stratified'`, `'residual'`, and `'multinomial'`. Defaults to `'default'`. Resampling methods other than `'default'` are currently experimental.

saveAll Indicates whether to save state samples for all time points (TRUE), or only for the most recent time point (FALSE)

smoothing Decides whether to save smoothed estimates of latent states, i.e., samples from f(x[1:t]|y[1:t]) if smoothing = TRUE, or instead to save filtered samples from f(x[t]|y[1:t]) if smoothing = FALSE. smoothing = TRUE only works if saveAll = TRUE.

timeIndex An integer used to manually specify which dimension of the latent state variable indexes time. This need only be set if the number of time points is less than or equal to the size of the latent state at each time point.

initModel A logical value indicating whether to initialize the model before running the filtering algorithm. Defaults to TRUE.

The auxiliary particle filter modifies the bootstrap filter (buildBootstrapFilter) by adding a lookahead step to the algorithm: before propagating particles from one time point to the next via the transition equation, the auxiliary filter calculates a weight for each pre-propogated particle by predicting how well the particle will agree with the next data point. These pre-weights are used to conduct an initial resampling step before propagation.

The resulting specialized particle filter algorthm will accept a single integer argument (m, default 10,000), which specifies the number of random \'particles\' to use for estimating the log-likelihood. The algorithm returns the estimated log-likelihood value, and saves unequally weighted samples from the posterior distribution of the latent states in the mvWSamples modelValues object, with corresponding logged weights in mvWSamples['wts',]. An equally weighted sample from the posterior can be found in the mvEWsamp modelValues object.

The auxiliary particle filter uses a lookahead function to select promising particles before propagation. This function can eithre be the expected value of the latent state at the next time point (lookahead = 'mean') or a simulation from the distribution of the latent state at the next time point (lookahead = 'simulate'), conditioned on the current particle.

@section returnESS() Method: Calling the returnESS() method of an auxiliary particle filter after that filter has been run() for a given model will return a vector of ESS (effective sample size) values, one value for each time point.

Author(s)

Nicholas Michaud

References

Pitt, M.K., and Shephard, N. (1999). Filtering via simulation: Auxiliary particle filters. *Journal of the American Statistical Association* 94(446): 590-599.

See Also

Other particle filtering methods: buildBootstrapFilter, buildEnsembleKF, buildIteratedFilter2(), buildLiuWestFilter

Examples

```
## For illustration only.
exampleCode <- nimbleCode({
```

```
  x0 ~ dnorm(0, var = 1)
  x[1] ~ dnorm(.8 * x0, var = 1)
  y[1] ~ dnorm(x[1], var = .5)
  for(t in 2:10){
    x[t] ~ dnorm(.8 * x[t-1], var = 1)
    y[t] ~ dnorm(x[t], var = .5)
  }
})

model <- nimbleModel(code = exampleCode, data = list(y = rnorm(10)),
                     inits = list(x0 = 0, x = rnorm(10)))
my_AuxF <- buildAuxiliaryFilter(model, 'x',
                control = list(saveAll = TRUE, lookahead = 'mean'))
## Now compile and run, e.g.,
## Cmodel <- compileNimble(model)
## Cmy_AuxF <- compileNimble(my_AuxF, project = model)
## logLik <- Cmy_AuxF$run(m = 1000)
## ESS <- Cmy_AuxF$returnESS()
## aux_X <- as.matrix(Cmy_AuxF$mvEWSamples, 'x')
```

`buildBootstrapFilter` *Create a bootstrap particle filter algorithm to estimate log-likelihood.*

Description

Create a bootstrap particle filter algorithm for a given NIMBLE state space model.

Usage

```
buildBootstrapFilter(model, nodes, control = list())
```

Arguments

`model`	A nimble model object, typically representing a state space model or a hidden Markov model.
`nodes`	A character vector specifying the stochastic latent model nodes over which the particle filter will stochastically integrate to estimate the log-likelihood function. All provided nodes must be stochastic. Can be one of three forms: a variable name, in which case all elements in the variable are taken to be latent (e.g., 'x'); an indexed variable, in which case all indexed elements are taken to be latent (e.g., 'x[1:100]' or 'x[1:100, 1:2]'); or a vector of multiple nodes, one per time point, in increasing time order (e.g., c("x[1:2, 1]", "x[1:2, 2]", "x[1:2, 3]", "x[1:2, 4]")).
`control`	A list specifying different control options for the particle filter. Options are described in the details section below.

Details

Each of the `control()` list options are described in detail here:

thresh A number between 0 and 1 specifying when to resample: the resampling step will occur when the effective sample size is less than `thresh` times the number of particles. Defaults to 0.8. Note that at the last time step, resampling will always occur so that the `mvEWsamples` `modelValues` contains equally-weighted samples.

resamplingMethod The type of resampling algorithm to be used within the particle filter. Can choose between `'default'` (which uses NIMBLE's `rankSample()` function), `'systematic'`, `'stratified'`, `'residual'`, and `'multinomial'`. Defaults to `'default'`. Resampling methods other than `'default'` are currently experimental.

saveAll Indicates whether to save state samples for all time points (TRUE), or only for the most recent time point (FALSE)

smoothing Decides whether to save smoothed estimates of latent states, i.e., samples from f(x[1:t]|y[1:t]) if `smoothing = TRUE`, or instead to save filtered samples from f(x[t]|y[1:t]) if `smoothing = FALSE`. `smoothing = TRUE` only works if `saveAll = TRUE`.

timeIndex An integer used to manually specify which dimension of the latent state variable indexes time. Only needs to be set if the number of time points is less than or equal to the size of the latent state at each time point.

initModel A logical value indicating whether to initialize the model before running the filtering algorithm. Defaults to TRUE.

The bootstrap filter starts by generating a sample of estimates from the prior distribution of the latent states of a state space model. At each time point, these particles are propagated forward by the model's transition equation. Each particle is then given a weight proportional to the value of the observation equation given that particle. The weights are used to draw an equally-weighted sample of the particles at this time point. The algorithm then proceeds to the next time point. Neither the transition nor the observation equations are required to be normal for the bootstrap filter to work.

The resulting specialized particle filter algorithm will accept a single integer argument (`m`, default 10,000), which specifies the number of random \'particles\' to use for estimating the log-likelihood. The algorithm returns the estimated log-likelihood value, and saves unequally weighted samples from the posterior distribution of the latent states in the `mvWSamples` modelValues object, with corresponding logged weights in `mvWSamples['wts',]`. An equally weighted sample from the posterior can be found in the `mvEWSamples` `modelValues` object.

Note that if the `thresh` argument is set to a value less than 1, resampling may not take place at every time point. At time points for which resampling did not take place, `mvEWSamples` will not contain equally weighted samples. To ensure equally weighted samples in the case that `thresh < 1`, we recommend resampling from `mvWSamples` at each time point after the filter has been run, rather than using `mvEWSamples`.

`returnESS()` Method

Calling the `returnESS()` method of a bootstrap filter after that filter has been `run()` for a given model will return a vector of ESS (effective sample size) values, one value for each time point.

Author(s)

Daniel Turek and Nicholas Michaud

References

Gordon, N.J., D.J. Salmond, and A.F.M. Smith. (1993). Novel approach to nonlinear/non-Gaussian Bayesian state estimation. *IEEE Proceedings F (Radar and Signal Processing)*. Vol. 140. No. 2. IET Digital Library, 1993.

See Also

Other particle filtering methods: buildAuxiliaryFilter, buildEnsembleKF, buildIteratedFilter2(), buildLiuWestFilter

Examples

```
## For illustration only.
exampleCode <- nimbleCode({
  x0 ~ dnorm(0, var = 1)
  x[1] ~ dnorm(.8 * x0, var = 1)
  y[1] ~ dnorm(x[1], var = .5)
  for(t in 2:10){
    x[t] ~ dnorm(.8 * x[t-1], var = 1)
    y[t] ~ dnorm(x[t], var = .5)
  }
})

model <- nimbleModel(code = exampleCode, data = list(y = rnorm(10)),
                     inits = list(x0 = 0, x = rnorm(10)))
my_BootF <- buildBootstrapFilter(model, 'x',
                control = list(saveAll = TRUE, thresh = 1))
## Now compile and run, e.g.,
## Cmodel <- compileNimble(model)
## Cmy_BootF <- compileNimble(my_BootF, project = model)
## logLik <- Cmy_BootF$run(m = 1000)
## ESS <- Cmy_BootF$returnESS()
## boot_X <- as.matrix(Cmy_BootF$mvEWSamples, 'x')
```

`buildEnsembleKF` *Create an Ensemble Kalman filter algorithm to sample from latent states.*

Description

Create an Ensemble Kalman filter algorithm for a given NIMBLE state space model.

Usage

```
buildEnsembleKF(model, nodes, control = list())
```

Arguments

model	A NIMBLE model object, typically representing a state space model or a hidden Markov model
nodes	A character vector specifying the stochastic latent model nodes the Ensemble Kalman Filter will estimate. All provided nodes must be stochastic. Can be one of three forms: a variable name, in which case all elements in the variable are taken to be latent (e.g., 'x'); an indexed variable, in which case all indexed elements are taken to be latent (e.g., 'x[1:100]' or 'x[1:100, 1:2]'); or a vector of multiple nodes, one per time point, in increasing time order (e.g., c("x[1:2, 1]", "x[1:2, 2]", "x[1:2, 3]", "x[1:2, 4]")).
control	A list specifying different control options for the particle filter. Options are described in the details section below.

Details

The control() list option is described in detail below:

saveAll Indicates whether to save state samples for all time points (TRUE), or only for the most recent time point (FALSE)

timeIndex An integer used to manually specify which dimension of the latent state variable indexes time. Only needs to be set if the number of time points is less than or equal to the size of the latent state at each time point.

initModel A logical value indicating whether to initialize the model before running the filtering algorithm. Defaults to TRUE.

Runs an Ensemble Kalman filter to estimate a latent state given observations at each time point. The ensemble Kalman filter is a Monte Carlo approximation to a Kalman filter that can be used when the model's transition euqations do not follow a normal distribution. Latent states (x[t]) and observations (y[t]) can be scalars or vectors at each time point, and sizes of observations can vary from time point to time point. In the BUGS model, the observations (y[t]) must be equal to some (possibly nonlinear) deterministic function of the latent state (x[t]) plus an additive error term. Currently only normal and multivariate normal error terms are supported. The transition from x[t] to x[t+1] does not have to be normal or linear. Output from the posterior distribution of the latent states is stored in mvSamples.

Author(s)

Nicholas Michaud

References

Houtekamer, P.L., and H.L. Mitchell. (1998). Data assimilation using an ensemble Kalman filter technique. *Monthly Weather Review*, 126(3), 796-811.

See Also

Other particle filtering methods: buildAuxiliaryFilter, buildBootstrapFilter, buildIteratedFilter2(), buildLiuWestFilter

Examples

```
## For illustration only.
exampleCode <- nimbleCode({
  x0 ~ dnorm(0, var = 1)
  x[1] ~ dnorm(.8 * x0, var = 1)
  y[1] ~ dnorm(x[1], var = .5)
  for(t in 2:10){
    x[t] ~ dnorm(.8 * x[t-1], var = 1)
    y[t] ~ dnorm(x[t], var = .5)
  }
})

model <- nimbleModel(code = exampleCode, data = list(y = rnorm(10)),
                     inits = list(x0 = 0, x = rnorm(10)))
my_enKF <- buildEnsembleKF(model, 'x',
                control = list(saveAll = TRUE, thresh = 1))
## Now compile and run, e.g.,
## Cmodel <- compileNimble(model)
## Cmy_enKF <- compileNimble(my_enKF, project = model)
## Cmy_enKF$run(m = 1000)
## enKF_X <- as.matrix(Cmy_enKF$mvSamples, 'x')
```

---

buildIteratedFilter2 *Create an IF2 algorithm.*

Description

Create an IF2 algorithm for a given NIMBLE state space model.

Usage

```
buildIteratedFilter2(
  model,
  nodes,
  params = NULL,
  baselineNode = NULL,
  control = list()
)
```

Arguments

model	A NIMBLE model object, typically representing a state space model or a hidden Markov model.
nodes	A character vector specifying the stochastic latent model nodes over which the particle filter will stochastically integrate to estimate the log-likelihood function. All provided nodes must be stochastic. Can be one of three forms: a variable name, in which case all elements in the variable are taken to be latent (e.g., 'x'); an indexed variable, in which case all indexed elements are taken to be

latent (e.g., 'x[1:100]' or 'x[1:100, 1:2]'); or a vector of multiple nodes, one per time point, in increasing time order (e.g., c("x[1:2, 1]", "x[1:2, 2]", "x[1:2, 3]", "x[1:2, 4]")).

`params` A character vector specifying the top-level parameters to obtain maximum likelihood estimates of. If unspecified, parameter nodes are specified as all stochastic top level nodes which are not in the set of latent nodes specified in `nodes`.

`baselineNode` A character vector specifying the node that is the latent node at the "0th" time step. The first node in `nodes` should depend on this baseline, but `baselineNode` should have no data depending on it. If `NULL` (the default), any initial state is taken to be fixed at the values present in the model at the time the algorithm is run.

`control` A list specifying different control options for the IF2 algorithm. Options are described in the 'details' section below.

Details

Each of the `control()` list options are described in detail below:

sigma A vector specifying a non-negative perturbation magnitude for each element of the `params` argument. Defaults to a vector of 1's.

initParamSigma An optional vector specifying a vector of standard deviations to use when simulating an initial particle swarm centered on the initial value of the parameters. Defaults to `sigma`.

inits A vector specifying an initial value for each element of the `params` argument. Defaults to the parameter values in the model at the time the model is built.

timeIndex An integer used to manually specify which dimension of the latent state variable indexes time. Only needs to be set if the number of time points is less than or equal to the size of the latent state at each time point.

initModel A logical value indicating whether to initialize the model before running the filtering algorithm. Defaults to `TRUE`.

The IF2 agorithm uses iterated filtering to estimate maximum likelihood values for top-level parameters for a state space model.

The resulting specialized IF2 algorithm will accept the following arguments:

m A single integer specifying the number of particles to use for each run of the filter.

n A single integer specifying the number of overall filter iterations to run.

alpha A double specifying the cooling factor to use for the IF2 algorithm.

The `run` fuction will return a vector with the estimated MLE. Additionally, once the specialized algorithm has been run, it can be continued for additional iterations by calling the `continueRun` method.

Reparameterization

The IF2 algorithm perturbs the parameters using a normal distribution, which may not be optimal for parameters whose support is not the whole real line, such as variance parameters, which are restricted to be positive. We recommend that users reparameterize the model in advance, e.g., writing variances and standard deviations on the log scale and probabilities on the logit scale. This requires specifying priors directly on the transformed parameters.

Parameter prior distributions

While NIMBLE's IF2 algorithm requires prior distributions on the parameters, the IF2 algorithm produces maximum likelihood estimates and does not directly use those prior distributions. We require the prior distributions to be stated only so that we can automatically determine which model nodes are the parameters. The IF2 algorithm also makes use of any bounds on the parameters.

Diagnostics and information stored in the algorithm object

The IF2 algorithm stores the estimated MLEs, one from each iteration, in `estimates`. It also stores standard deviation of the particles from each iteration, one per parameter, in `estSD`. Finally it stores the estimated log-likelihood at the estimated MLE from each iteration in `logLik`.

Author(s)

Nicholas Michaud, Dao Nguyen, and Christopher Paciorek

References

Ionides, E.L., D. Nguyen, Y. Atchadé, S. Stoev, and A.A. King (2015). Inference for dynamic and latent variable models via iterated, perturbed Bayes maps. *Proceedings of the National Academy of Sciences*, 112(3), 719-724.

See Also

Other particle filtering methods: `buildAuxiliaryFilter`, `buildBootstrapFilter`, `buildEnsembleKF`, `buildLiuWestFilter`

Examples

```
## For illustration only.
exampleCode <- nimbleCode({
  x0 ~ dnorm(0, var = 1)
  x[1] ~ dnorm(.8 * x0, var = 1)
  y[1] ~ dnorm(x[1], var = .5)
  for(t in 2:10){
    x[t] ~ dnorm(.8 * x[t-1], sd = sigma_x)
    y[t] ~ dnorm(x[t], var = .5)
  }
  sigma_x ~ dunif(0, 10)
})

model <- nimbleModel(code = exampleCode, data = list(y = rnorm(10)),
```

```
                    inits = list(x0 = 0, x = rnorm(10), sigma_x = 1))
my_IF2 <- buildIteratedFilter2(model, 'x', params = 'sigma_x')
## Now compile and run, e.g.,
## Cmodel <- compileNimble(model)
## Cmy_IF2 <- compileNimble(my_IF2, project = model)
## MLE estimate of a top level parameter named sigma_x:
## sigma_x_MLE <- Cmy_IF2$run(m = 10000, n = 50, alpha = 0.2)
## visualize progression of the estimated log-likelihood
## ts.plot(Cmy_IF2$logLik)
## Continue running algorithm for more precise estimate:
## sigma_x_MLE <- Cmy_IF2$continueRun(n = 50, alpha = 0.2)
```

`buildLiuWestFilter` *Create a Liu and West particle filter algorithm.*

Description

Create a Liu and West particle filter algorithm for a given NIMBLE state space model.

Usage

```
buildLiuWestFilter(model, nodes, params = NULL, control = list())
```

Arguments

`model`	A NIMBLE model object, typically representing a state space model or a hidden Markov model
`nodes`	A character vector specifying the stochastic latent model nodes over which the particle filter will stochastically integrate to estimate the log-likelihood function. All provided nodes must be stochastic. Can be one of three forms: a variable name, in which case all elements in the variable are taken to be latent (e.g., 'x'); an indexed variable, in which case all indexed elements are taken to be latent (e.g., 'x[1:100]' or 'x[1:100, 1:2]'); or a vector of multiple nodes, one per time point, in increasing time order (e.g., c("x[1:2, 1]", "x[1:2, 2]", "x[1:2, 3]", "x[1:2, 4]")).
`params`	A character vector specifying the top-level parameters to estimate the posterior distribution of. If unspecified, parameter nodes are specified as all stochastic top level nodes which are not in the set of latent nodes specified in `nodes`.
`control`	A list specifying different control options for the particle filter. Options are described in the details section below.

Details

Each of the `control()` list options are described in detail below:

d A discount factor for the Liu-West filter. Should be close to, but not above, 1.

saveAll Indicates whether to save state samples for all time points (TRUE), or only for the most recent time point (FALSE)

timeIndex An integer used to manually specify which dimension of the latent state variable indexes time. Only needs to be set if the number of time points is less than or equal to the size of the latent state at each time point.

initModel A logical value indicating whether to initialize the model before running the filtering algorithm. Defaults to TRUE.

The Liu and West filter samples from the posterior distribution of both the latent states and top-level parameters for a state space model. Each particle in the Liu and West filter contains values not only for latent states, but also for top level parameters. Latent states are propogated via an auxiliary step, as in the auxiliary particle filter (buildAuxiliaryFilter). Top-level parameters are propagated from one time point to the next through a smoothed kernel density based on previous particle values.

The resulting specialized particle filter algorthm will accept a single integer argument (m, default 10,000), which specifies the number of random \'particles\' to use for sampling from the posterior distributions. The algorithm saves unequally weighted samples from the posterior distribution of the latent states and top-level parameters in mvWSamples, with corresponding logged weights in mvWSamples['wts',]. An equally weighted sample from the posterior can be found in mvEWSamples.

Note that if saveAll=TRUE, the top-level parameter samples given in the mvWSamples output will correspond to the weights from the final time point.

Author(s)

Nicholas Michaud

References

Liu, J., and M. West. (2001). Combined parameter and state estimation in simulation-based filtering. *Sequential Monte Carlo methods in practice*. Springer New York, pages 197-223.

See Also

Other particle filtering methods: buildAuxiliaryFilter, buildBootstrapFilter, buildEnsembleKF, buildIteratedFilter2()

Examples

```
## For illustration only.
exampleCode <- nimbleCode({
  x0 ~ dnorm(0, var = 1)
  x[1] ~ dnorm(.8 * x0, var = 1)
  y[1] ~ dnorm(x[1], var = .5)
  for(t in 2:10){
    x[t] ~ dnorm(.8 * x[t-1], sd = sigma_x)
    y[t] ~ dnorm(x[t], var = .5)
  }
  sigma_x ~ dunif(0, 10)
})

model <- nimbleModel(code = exampleCode, data = list(y = rnorm(10)),
```

```
                    inits = list(x0 = 0, x = rnorm(10), sigma_x = 1))
my_LWF <- buildLiuWestFilter(model, 'x', params = 'sigma_x')
## Now compile and run, e.g.,
## Cmodel <- compileNimble(model)
## Cmy_LWF <- compileNimble(my_LWF, project = model)
## Cmy_LWF$run(m = 1000)
## lw_X <- as.matrix(Cmy_LWF$mvEWSamples, 'x')
##  samples from posterior of top level parameter
## lw_sigma_x <- as.matrix(Cmy_LWF$mvEWSamples, 'sigma_x')
```

SMCsamplers *Particle Filtering MCMC Sampling Algorithms*

Description

Details of the particle filtering MCMC sampling algorithms provided in nimbleSMC.

Usage

```
sampler_RW_PF(model, mvSaved, target, control)

sampler_RW_PF_block(model, mvSaved, target, control)
```

Arguments

model	(uncompiled) model on which the MCMC is to be run
mvSaved	modelValues object to be used to store MCMC samples
target	node(s) on which the sampler will be used
control	named list that controls the precise behavior of the sampler, with elements specific to samplertype. The default values for control list are specified in the setup code of each sampling algorithm. Descriptions of each sampling algorithm, and the possible customizations for each sampler (using the control argument) appear below.

RW_PF sampler

The particle filter sampler allows the user to perform particle MCMC (PMCMC) (Andrieu et al., 2010), primarily for state-space or hidden Markov models of time-series data. This method uses Metropolis-Hastings samplers for top-level parameters but uses the likelihood approximation of a particle filter (sequential Monte Carlo) to integrate over latent nodes in the time-series. The RW_PF sampler uses an adaptive Metropolis-Hastings algorithm with a univariate normal proposal distribution for a scalar parameter. Note that samples of the latent states can be retained as well, but the top-level parameter being sampled must be a scalar. A bootstrap, auxiliary, or user defined particle filter can be used to integrate over latent states.

For more information about user-defined samplers within a PMCMC sampler, see the NIMBLE User Manual.

The RW_PF sampler accepts the following control list elements:

- adaptive. A logical argument, specifying whether the sampler should adapt the scale (proposal standard deviation) throughout the course of MCMC execution to achieve a theoretically desirable acceptance rate. (default = TRUE)
- adaptInterval. The interval on which to perform adaptation. Every adaptInterval MCMC iterations (prior to thinning), the RW sampler will perform its adaptation procedure. This updates the scale variable, based upon the sampler's achieved acceptance rate over the past adaptInterval iterations. (default = 200)
- scale. The initial value of the normal proposal standard deviation. If `adaptive = FALSE`, scale will never change. (default = 1)
- pfNparticles. The number of particles to use in the approximation to the log likelihood of the data (default = 1000).
- latents. Character vector specifying the nodes that are latent states over which the particle filter will operate to approximate the log-likelihood function.
- pfType. Character argument specifying the type of particle filter that should be used for likelihood approximation. Choose from `"bootstrap"` and `"auxiliary"`. Defaults to `"bootstrap"`.
- pfControl. A control list that is passed to the particle filter function. For details on control lists for bootstrap or auxiliary particle filters, see `buildBootstrapFilter` or `buildAuxiliaryFilter` respectively. Additionally, this can be used to pass custom arguments into a user-defined particle filter.
- pfOptimizeNparticles. A logical argument, specifying whether to use an experimental procedure to automatically determine the optimal number of particles to use, based on Pitt and Shephard (2011). This will override any value of `pfNparticles` specified above.
- pf. A user-defined particle filter object, if a bootstrap or auxiliary particle filter is not adequate.

RW_PF_block sampler

The particle filter block sampler allows the user to perform particle MCMC (PMCMC) (Andrieu et al., 2010) for multiple parameters jointly, primarily for state-space or hidden Markov models of time-series data. This method uses Metropolis-Hastings block samplers for top-level parameters but uses the likelihood approximation of a particle filter (sequential Monte Carlo) to integrate over latent nodes in the time-series. The `RW_PF` sampler uses an adaptive Metropolis-Hastings algorithm with a multivariate normal proposal distribution. Note that samples of the latent states can be retained as well, but the top-level parameter being sampled must be a scalar. A bootstrap, auxiliary, or user defined particle filter can be used to integrate over latent states.

For more information about user-defined samplers within a PMCMC sampler, see the NIMBLE User Manual.

The `RW_PF_block` sampler accepts the following control list elements:

- adaptive. A logical argument, specifying whether the sampler should adapt the proposal covariance throughout the course of MCMC execution. (default = TRUE)
- adaptScaleOnly. A logical argument, specifying whether adaptation should be done only for `scale` (TRUE) or also for `provCov` (FALSE). This argument is only relevant when `adaptive = TRUE`. When `adaptScaleOnly = FALSE`, both `scale` and `propCov` undergo adaptation; the sampler tunes the scaling to achieve a theoretically good acceptance rate, and the proposal covariance to mimic that of the empirical samples. When `adaptScaleOnly = TRUE`, only the proposal scale is adapted. (default = FALSE)

- adaptInterval. The interval on which to perform adaptation. (default = 200)
- scale. The initial value of the scalar multiplier for `propCov`. If `adaptive = FALSE`, `scale` will never change. (default = 1)
- adaptFactorExponent. Exponent controling the rate of decay of the scale adaptation factor. See Shaby and Wells, 2011, for details. (default = 0.8)
- propCov. The initial covariance matrix for the multivariate normal proposal distribution. This element may be equal to the `'identity'`, in which case the identity matrix of the appropriate dimension will be used for the initial proposal covariance matrix. (default is `'identity'`)
- pfNparticles. The number of particles to use in the approximation to the log likelihood of the data (default = 1000).
- latents. Character vector specifying the nodes that are latent states over which the particle filter will operate to approximate the log-likelihood function.
- pfType. Character argument specifying the type of particle filter that should be used for likelihood approximation. Choose from `"bootstrap"` and `"auxiliary"`. Defaults to `"bootstrap"`.
- pfControl. A control list that is passed to the particle filter function. For details on control lists for bootstrap or auxiliary particle filters, see `buildBootstrapFilter` or `buildAuxiliaryFilter` respectively. Additionally, this can be used to pass custom arguments into a user defined particle filter.
- pfOptimizeNparticles. A logical argument, specifying whether to automatically determine the optimal number of particles to use, based on Pitt and Shephard (2011). This will override any value of `pfNparticles` specified above.
- pf. A user-defined particle filter object, if a bootstrap or auxiliary particle filter is not adequate.

Index