Package ‘fmcmc’

April 23, 2020

Title A friendly MCMC framework

Version 0.3-0

Description Provides a friendly (flexible) Markov Chain Monte Carlo (MCMC) framework for implementing Metropolis-Hastings algorithm in a modular way allowing users to specify automatic convergence checker, personalized transition kernels, and out-of-the-box multiple MCMC chains using parallel computing. Most of the methods implemented in this package can be found in Brooks et al. (2011, ISBN 9781420079425). Among the methods included, we have: Haario (2001) <doi:10.1007/s11222-011-9269-5> Adaptive Metropolis, Vihola (2012) <doi:10.1007/s11222-011-9269-5> Robust Adaptive Metropolis, and Thawornwattana et al. (2018) <doi:10.1214/17-BA1084> Mirror transition kernels.

Depends R (>= 3.3.0)

License MIT + file LICENSE

Encoding UTF-8

Language en-US

LazyData true

URL `https://github.com/USCbiostats/fmcmc`

BugReports `https://github.com/USCbiostats/fmcmc/issues`

Suggests covr, knitr, rmarkdown, mcmc, tinytest, mvtnorm, adaptMCMC

Imports parallel, coda, stats, methods, MASS

RoxygenNote 7.0.2

VignetteBuilder knitr

NeedsCompilation no

Author George Vega Yon [aut, cre] (<https://orcid.org/0000-0002-3171-0844>), Paul Marjoram [ctb, ths] (<https://orcid.org/0000-0003-0824-7449>), National Cancer Institute (NCI) [fnd] (Grant Number 5P01CA196569-02), Fabian Scheipl [rev] (JOSS reviewer, <https://orcid.org/0000-0001-8172-3603>)

Maintainer George Vega Yon <`g.vegayon@gmail.com`>

Repository CRAN

Date/Publication 2020-04-23 00:14:15 UTC

R topics documented:

`append_chains` *Append MCMC chains (objects of class coda::mcmc)*

Description

Combines two or more MCMC runs into a single run. If runs have multiple chains, it will check that all have the same number of chains, and it will join chains using the rbind function.

Usage

```
append_chains(...)

## Default S3 method:
append_chains(...)

## S3 method for class 'mcmc.list'
append_chains(...)

## S3 method for class 'mcmc'
append_chains(...)
```

Arguments

`...`	A list of `mcmc` or `mcmc.list` class objects.

Value

If `mcmc.list`, an object of class `mcmc.list`, otherwise, an object of class `mcmc`.

check_initial *Checks the initial values of the MCMC*

Description

This function is for internal use only.

Usage

```
check_initial(initial, nchains)
```

Arguments

initial	Either a vector or matrix,.
nchains	Integer scalar. Number of chains.

Details

When `initial` is a vector, the values are recycled to form a matrix of size `nchains * length(initial)`.

Value

A named matrix.

Examples

```
init <- c(.4, .1)
check_initial(init, 1)
check_initial(init, 2)

init <- matrix(1:9, ncol=3)
check_initial(init, 3)

# check_initial(init, 2) # Returns an error
```

convergence-checker *Convergence Monitoring*

Description

Built-in set of functions to be used in companion with the argument `conv_checker` in MCMC. These functions are not intended to be used in a context other than the `MCMC` function.

Usage

```
convergence_gelman(freq = 1000L, threshold = 1.1, check_invariant = TRUE, ...)

convergence_geweke(
  freq = 1000L,
  threshold = 0.025,
  check_invariant = TRUE,
  ...
)

convergence_heildel(freq = 1000L, ..., check_invariant = TRUE)

convergence_auto(freq = 1000L)
```

Arguments

`freq`	Integer scalar. Frequency of checking.
`threshold`	Numeric value. A Gelman statistic below the threshold will return `TRUE`.
`check_invariant`	Logical. When `TRUE` the function only computes the Gelman diagnostic using variables with greater than `1e-10` variance.
`...`	Further arguments passed to the method.

Details

In the case of `convergence_geweke`, `threshold` sets the p-value for the null $H_0 : Z = 0$, i.e. equal means between the first and last chunks of the chain. See coda::geweke.diag. This implies that the higher the threshold, the lower the probability of stopping the chain.

In the case that the chain has more than one parameter, the algorithm will return true if and only if the test fails to reject the null for all the parameters.

For the `convergence_heildel`, see coda::heidel.diag for details.

The `convergence_auto` function is the default and is just a wrapper for `convergence_gelman` and `convergence_geweke`. This function returns a convergence checker that will be either of the other two depending on whether `nchains` in `MCMC` is greater than one–in which case it will use the Gelman test–or not–in which case it will use the Geweke test.

Value

A function passed to MCMC to check automatic convergence.

cov_recursive *Recursive algorithms for computing variance and mean*

Description

These algorithms are used in `kernel_adapt()` to simplify variance-covariance recalculation at every step of the algorithm.

Usage

```
cov_recursive(
  X_t,
  Cov_t,
  Mean_t,
  Mean_t_prev,
  t.,
  eps = 0,
  Sd = 1,
  Ik = diag(length(X_t))
)

mean_recursive(X_t, Mean_t_prev, t.)
```

Arguments

`X_t`	Last value of the sample
`Cov_t`	Covariance in t
`Mean_t, Mean_t_prev`	Vectors of averages in time `t` and `t-1` respectively.
`t.`	Sample size up to `t-1`.
`Sd, eps, Ik`	See `kernel_adapt()`.

Details

The variance covariance algorithm was described in Haario, Saksman and Tamminen (2002).

References

Haario, H., Saksman, E., & Tamminen, J. (2001). An adaptive Metropolis algorithm. Bernoulli, 7(2), 223–242. https://projecteuclid.org/euclid.bj/1080222083

Examples

```
# Generating random data (only four points to see the difference)
set.seed(1231)
n <- 3
X <- matrix(rnorm(n*4), ncol = 4)
```

```
# These two should be equal
mean_recursive(
  X_t         = X[1,],
  Mean_t_prev = colMeans(X[-1,]),
  t.          = n - 1
)
colMeans(X)

# These two should be equal
cov_recursive(
  X_t         = X[1, ],
  Cov_t       = cov(X[-1,]),
  Mean_t      = colMeans(X),
  Mean_t_prev = colMeans(X[-1, ]),
  t           = n-1
)
cov(X)

# Speed example -------------------------------------------------------------
set.seed(13155511)
X <- matrix(rnorm(1e3*100), ncol = 100)

ans0 <- cov(X[-1,])
t0 <- system.time({
  ans1 <- cov(X)
})

t1 <- system.time(ans2 <- cov_recursive(
  X[1, ], ans0,
  Mean_t      = colMeans(X),
  Mean_t_prev = colMeans(X[-1,]),
  t. = 1e3 - 1
))

# Comparing accuracy and speed
range(ans1 - ans2)
t0/t1
```

fmcmc *A friendly MCMC framework*

Description

The fmcmc package provides a flexible framework for implementing MCMC models using a lightweight in terms of dependencies. Among its main features, fmcmc allows:

Details

- Implementing arbitrary transition kernels.
- Incorporating convergence monitors for automatic stop.
- Out-of-the-box parallel computing implementation for running multiple chains simultaneously.

For more information see the packages vignettes:

```
vignette("workflow-with-fmcmc", "fmcmc")
```

```
vignette("user-defined-kernels", "fmcmc")
```

References

Vega Yon et al., (2019). fmcmc: A friendly MCMC framework. Journal of Open Source Software, 4(39), 1427, doi: 10.21105/joss.01427

`kernel_adapt` *Adaptive Metropolis (AM) Transition Kernel*

Description

Implementation of Haario et al. (2001)'s Adaptive Metropolis.

Usage

```
kernel_adapt(
  mu = 0,
  bw = 0L,
  lb = -.Machine$double.xmax,
  ub = .Machine$double.xmax,
  freq = 1L,
  warmup = 500L,
  Sigma = NULL,
  Sd = NULL,
  eps = 1e-04,
  fixed = FALSE,
  until = Inf
)

kernel_am(
  mu = 0,
  bw = 0L,
  lb = -.Machine$double.xmax,
  ub = .Machine$double.xmax,
  freq = 1L,
  warmup = 500L,
  Sigma = NULL,
  Sd = NULL,
  eps = 1e-04,
  fixed = FALSE,
  until = Inf
)
```

Arguments

`mu`	Either a numeric vector or a scalar. Proposal mean. If scalar, values are recycled to match the number of parameters in the objective function.
`bw`	Integer scalar. The bandwidth, is the number of observations to include in the computation of the variance-covariance matrix.
`lb, ub`	Either a numeric vector or a scalar. Lower and upper bounds for bounded kernels. When of length 1, the values are recycled to match the number of parameters in the objective function.
`freq`	Integer scalar. Frequency of updates. How often the variance-covariance matrix is updated. The implementation is different from that described in the original paper (see details).
`warmup`	Integer scalar. The number of iterations that the algorithm has to wait before starting to do the updates.
`Sigma`	The variance-covariance matrix. By default this will be an identity matrix during the warmup period.
`Sd`	Overall scale for the algorithm. By default, the variance-covariance is scaled to $2.4^2/d$, with d the number of dimensions.
`eps`	Double scalar. Default size of the initial step (see details).
`fixed`	Logical scalar or vector of length `k`. Indicates which parameters will be treated as fixed or not. Single values are recycled.
`until`	Integer scalar. Last step at which adaptation takes place (see details).

Details

While it has been shown that under regular conditions this transition kernel generates ergodic chains even when the adaptation does not stop, some practitioners may want to stop adaptation at some point.

`kernel_adapt` Implements the adaptive Metropolis (AM) algorithm of Haario et al. (2001). If the value of bw is greater than zero, then the algorithm folds back AP, a previous version which is known to have ergodicity problems.

The parameter `eps` has two functions. The first one is to set the initial scale for the multivariate normal kernel, which is replaced after `warmup` steps with the actual variance-covariance computed by the main algorithm. The second usage is in the equation that ensures that the variance-covariance is greater than zero, this is, the ε parameter in the original paper.

The update of the covariance matrix is done using `cov_recursive()` function, which makes the updates faster. The `freq` parameter, besides of indicating the frequency with which the updates are done, it specifies what are the samples included in each update, in other words, like a thinning parameter, only every `freq` samples will be used to compute the covariance matrix. Since this implementation uses the recursive formula for updating the covariance, there is no practical need to set `freq != 1`.

`kernel_am` is just an alias for `kernel_adapt`.

Value

An object of class fmcmc_kernel. `fmcmc_kernel` objects are intended to be used with the `MCMC()` function.

References

Haario, H., Saksman, E., & Tamminen, J. (2001). An adaptive Metropolis algorithm. Bernoulli, 7(2), 223–242. `https://projecteuclid.org/euclid.bj/1080222083`

See Also

Other kernels: `kernel_mirror`, `kernel_new()`, `kernel_normal()`, `kernel_ram()`, `kernel_unif()`

Examples

```
# Update every-step and wait 1,000 steps before starting to adapt
kern <- kernel_adapt(freq = 1, warmup = 1000)

# Two parameters model, the second parameter with a restricted range, i.e.
# a lower bound of 1
kern <- kernel_adapt(lb = c(-.Machine$double.xmax, 0))
```

`kernel_mirror` *Mirror Transition Kernels*

Description

NMirror and UMirror transition kernels described in Thawornwattana et al. (2018).

Usage

```
kernel_nmirror(
  mu = 0,
  scale = 1,
  warmup = 500L,
  nadapt = 4L,
  arate = 0.4,
  lb = -.Machine$double.xmax,
  ub = .Machine$double.xmax,
  fixed = FALSE,
  scheme = "joint"
)
```

```
kernel_umirror(
  mu = 0,
  scale = 1,
  warmup = 500L,
  nadapt = 4L,
  arate = 0.4,
  lb = -.Machine$double.xmax,
  ub = .Machine$double.xmax,
  fixed = FALSE,
  scheme = "joint"
)
```

Arguments

`mu, scale`	Either a numeric vector or a scalar. Proposal mean and scale. If scalar, values are recycled to match the number of parameters in the objective function.
`warmup`	Integer. Number of steps required before starting adapting the chains.
`nadapt`	Integer. Number of times the scale is adjusted for adaptation during the warmup (burn-in) period.
`arate`	Double. Target acceptance rate used as a reference during the adaptation process.
`lb, ub`	Either a numeric vector or a scalar. Lower and upper bounds for bounded kernels. When of length 1, the values are recycled to match the number of parameters in the objective function.
`fixed, scheme`	For multivariate functions, sets the update plan. See `plan_update_sequence()`.

Details

The `kernel_nmirror` and `kernel_umirror` functions implement simple symmetric transition kernels that pivot around an approximation of the asymptotic mean.

In the multidimensional case, this implementation just draws a vector of independent draws from the proposal kernel, instead of using, for example, a multivariate distribution of some kind. This will be implemented in the next update of the package.

During the warmup period (or burnin as described in the paper), the algorithm adapts both the scale and the reference mean of the proposal distribution. While the mean is adapted continuously, the scale is updated only a handful of times, in particular, `nadapt` times during the warmup time. The adaptation is done as proposed by Yang and Rodriguez (2013) in which the scale is adapted four times.

Value

An object of class fmcmc_kernel. `fmcmc_kernel` objects are intended to be used with the `MCMC()` function.

References

Thawornwattana, Y., Dalquen, D., & Yang, Z. (2018). Designing Simple and Efficient Markov Chain Monte Carlo Proposal Kernels. Bayesian Analysis, 13(4), 1037–1063. https://doi.org/10.1214/17-BA1084

Yang, Z., & Rodriguez, C. E. (2013). Searching for efficient Markov chain Monte Carlo proposal kernels. Proceedings of the National Academy of Sciences, 110(48), 19307–19312. https://doi.org/10.1073/pnas.1311790110

See Also

Other kernels: kernel_adapt(), kernel_new(), kernel_normal(), kernel_ram(), kernel_unif()

Examples

```
# Normal mirror kernel with 5 adaptations and 1000 steps of warmup (burnin)
kern <- kernel_nmirror(nadapt = 5, warmup = 1000)

# Same as before but using a uniform mirror and choosing a target acceptance
# rate of 24 %
kern <- kernel_umirror(nadapt = 5, warmup = 1000, arate = .24)
```

---

kernel_new *Transition Kernels for MCMC*

Description

The function kernel_new is a helper function that allows creating fmcmc_kernel objects which are passed to the MCMC() function.

Usage

```
kernel_new(proposal, ..., logratio = NULL, kernel_env = new.env(hash = TRUE))
```

Arguments

proposal, logratio	Functions. Both receive a single argument, an environment. This functions are called later within MCMC (see details).
...	In the case of kernel_new, further arguments to be stored with the kernel.
kernel_env	Environment. This will be used as the main container of the kernel's components. It is returned as an object of class c("environment","fmcmc_kernel").

Details

The objects `fmcmc_kernels` are environments that in general contain the following objects:

- `proposal`: The function used to propose changes in the chain based on the current state. The function must return a vector of length equal to the number of parameters in the model.
- `logratio`: This function is called after a new state has been proposed, and is used to compute the log of the Hastings ratio.

  In the case that the `logratio` function is not specified, then it is assumed that the transition kernel is symmetric, this is, log-ratio is then implemented as `function(env) {env$f1 -env$f0}`
- `...`: Further objects that are used within those functions.

Both functions, `proposal` and `logratio`, receive a single argument, an environment, which is passed by the `MCMC()` function during each step using the function `environment()`.

The passed environment is actually the environment in which the `MCMC` function is running, in particular, this environment contains the following objects:

Object	**Description**
`i`	Integer. The current iteration.
`theta1`	Numeric vector. The last proposed state.
`theta0`	Numeric vector. The current state
`f`	The log-unnormalized posterior function (a wrapper of `fun` passed to MCMC).
`f1`	The last value of `f(theta1)`
`f0`	The last value of `f(theta0)`
`kernel`	The actual `fmcmc_kernel` object.
`ans`	The matrix of samples defined up to `i - 1`.

These are the core component of the `MCMC` function. The following block of code is how this is actually implemented in the package:

```
for (i in 1L:nsteps) {
  # Step 1. Propose
  theta1[] <- kernel$proposal(environment())
  f1       <- f(theta1)

  # Checking f(theta1) (it must be a number, can be Inf)
  if (is.nan(f1) | is.na(f1) | is.null(f1))
    stop(
      "fun(par) is undefined (", f1, ")",
      "Check either -fun- or the -lb- and -ub- parameters.",
      call. = FALSE
    )

  # Step 2. Hastings ratio
  if (R[i] < kernel$logratio(environment())) {
    theta0 <- theta1
    f0     <- f1
  }

  # Step 3. Saving the state
  ans[i,] <- theta0

}
```

For an extensive example on how to create new kernel objects see the vignette `vignette("user-defined-kernels","fmcmc`

Value

An environment of class `fmcmc_kernel` which contains the following:

- `proposal` A function that receives a single argument, an environment. This is the proposal function used within `MCMC()`.
- `logratio` A function to compute log ratios of the current vs the proposed step of the chain. Also used within `MCMC()`.
- `...` Further arguments passed to `kernel_new`.

Behavior

In some cases, calls to the `proposal()` and `logratio()` functions in `fmcmc_kernels` can trigger changes or updates of variables stored within them. A concrete example is with adaptive kernels.

Adaptive Metropolis and Robust Adaptive Metropolis implemented in the functions `kernel_adapt()` and `kernel_ram()` both update a covariance matrix used during the proposal stage, and furthermore, have a `warmup` stage that sets the point at which both will start adapting. Because of this, both kernels have internal counters of the absolute step count which allows them activating, scaling, etc. the proposals correctly.

1. When running multiple chains, `MCMC` will create independent copies of a baseline passed `fmcmc_kernel` object. These are managed together in a `fmcmc_kernel_list` object.
2. Even if the chains are run in parallel, if a predefined kernel object is passed it will be updated to reflect the last state of the kernels before the `MCMC` call returns.

References

Brooks, S., Gelman, A., Jones, G. L., & Meng, X. L. (2011). Handbook of Markov Chain Monte Carlo. Handbook of Markov Chain Monte Carlo.

See Also

Other kernels: `kernel_adapt()`, `kernel_mirror`, `kernel_normal()`, `kernel_ram()`, `kernel_unif()`

Examples

```
# Example creating a multivariate normal kernel using the mvtnorm R package
# for a bivariate normal distribution
library(mvtnorm)
```

```
# Define your Sigma
sigma <- matrix(c(1, .2, .2, 1), ncol = 2)

# How does it looks like?
sigma
#      [,1] [,2]
# [1,]  1.0  0.2
# [2,]  0.2  1.0

# Create the kernel
kernel_mvn <- kernel_new(
  proposal = function(env) {
  env$theta0 + as.vector(mvtnorm::rmvnorm(1, mean = 0, sigma = sigma.))
  },
  sigma. = sigma
)

# As you can see, in the previous call we passed sigma as it will be used by
# the proposal function
# The logaratio function was not necesary to be passed since this kernel is
# symmetric.
```

kernel_normal *Gaussian Transition Kernel*

Description

Gaussian Transition Kernel

Usage

```
kernel_normal(mu = 0, scale = 1, fixed = FALSE, scheme = "joint")

kernel_normal_reflective(
  mu = 0,
  scale = 1,
  lb = -.Machine$double.xmax,
  ub = .Machine$double.xmax,
  fixed = FALSE,
  scheme = "joint"
)
```

Arguments

`mu, scale`	Either a numeric vector or a scalar. Proposal mean and scale. If scalar, values are recycled to match the number of parameters in the objective function.
`fixed, scheme`	For multivariate functions, sets the update plan. See `plan_update_sequence()`.

`lb, ub`	Either a numeric vector or a scalar. Lower and upper bounds for bounded kernels. When of length 1, the values are recycled to match the number of parameters in the objective function.

Details

The `kernel_normal` function provides the canonical normal kernel with symmetric transition probabilities.

The `kernel_normal_reflective` implements the normal kernel with reflective boundaries. Lower and upper bounds are treated using reflecting boundaries, this is, if the proposed θ' is greater than the `ub`, then $\theta' - ub$ is subtracted from ub. At the same time, if it is less than `lb`, then $lb - \theta'$ is added to `lb` iterating until θ is within `[lb,ub]`.

In this case, the transition probability is symmetric (just like the normal kernel).

Value

An object of class fmcmc_kernel. `fmcmc_kernel` objects are intended to be used with the `MCMC()` function.

See Also

Other kernels: `kernel_adapt`(), `kernel_mirror`, `kernel_new`(), `kernel_ram`(), `kernel_unif`()

Examples

```
# Normal kernel with a small scale (sd) of 0.05
kern <- kernel_normal(scale = 0.05)

# Using boundaries for the second parameter of a two parameter chain
# to have values in [0, 100].
kern <- kernel_normal_reflective(
  ub = c(.Machine$double.xmax, 100),
  lb = c(-.Machine$double.xmax, 0)
  )
```

`kernel_ram` *Robust Adaptive Metropolis (RAM) Transition Kernel*

Description

Implementation of Vihola (2012)'s Robust Adaptive Metropolis.

Usage

```
kernel_ram(
  mu = 0,
  eta = function(i, k) min(c(1, i^(-2/3) * k)),
  qfun = function(k) stats::rt(k, k),
  arate = 0.234,
  freq = 1L,
  warmup = 0L,
  Sigma = NULL,
  eps = 1e-04,
  lb = -.Machine$double.xmax,
  ub = .Machine$double.xmax,
  fixed = FALSE,
  until = Inf
)
```

Arguments

`mu`	Either a numeric vector or a scalar. Proposal mean. If scalar, values are recycled to match the number of parameters in the objective function.
`eta`	A function that receives the MCMC environment. This is to calculate the scaling factor for the adaptation.
`qfun`	Function. As described in Vihola (2012)'s, the `qfun` function is a symmetric function used to generate random numbers.
`arate`	Numeric scalar. Objective acceptance rate.
`freq`	Integer scalar. Frequency of updates. How often the variance-covariance matrix is updated.
`warmup`	Integer scalar. The number of iterations that the algorithm has to wait before starting to do the updates.
`Sigma`	The variance-covariance matrix. By default this will be an identity matrix during the warmup period.
`eps`	Double scalar. Default size of the initial step (see details).
`lb, ub`	Either a numeric vector or a scalar. Lower and upper bounds for bounded kernels. When of length 1, the values are recycled to match the number of parameters in the objective function.
`fixed`	Logical scalar or vector of length `k`. Indicates which parameters will be treated as fixed or not. Single values are recycled.
`until`	Integer scalar. Last step at which adaptation takes place (see details).

Details

While it has been shown that under regular conditions this transition kernel generates ergodic chains even when the adaptation does not stop, some practitioners may want to stop adaptation at some point.

The idea is similar to that of the Adaptive Metropolis algorithm (AM implemented as `kernel_adapt()` here) with the difference that it takes into account a target acceptance rate.

The `eta` function regulates the rate of adaptation. The default implementation will decrease the rate of adaptation exponentially as a function of the iteration number.

Value

An object of class fmcmc_kernel.

References

Vihola, M. (2012). Robust adaptive Metropolis algorithm with coerced acceptance rate. Statistics and Computing, 22(5), 997–1008. `https://doi.org/10.1007/s11222-011-9269-5`

See Also

Other kernels: `kernel_adapt()`, `kernel_mirror`, `kernel_new()`, `kernel_normal()`, `kernel_unif()`

Examples

```
# Setting the acceptance rate to 30 % and deferring the updates until
# after 1000 steps
kern <- kernel_ram(arate = .3, warmup = 1000)
```

`kernel_unif` *Uniform Transition Kernel*

Description

Uniform Transition Kernel

Usage

```
kernel_unif(min. = -1, max. = 1, fixed = FALSE, scheme = "joint")

kernel_unif_reflective(
  min. = -1,
  max. = 1,
  lb = min.,
  ub = max.,
  fixed = FALSE,
  scheme = "joint"
)
```

Arguments

`min., max.`	Passed to `stats::runif()`.
`fixed, scheme`	For multivariate functions, sets the update plan. See `plan_update_sequence()`.
`lb, ub`	Either a numeric vector or a scalar. Lower and upper bounds for bounded kernels. When of length 1, the values are recycled to match the number of parameters in the objective function.

Details

The `kernel_unif` function provides a uniform transition kernel. This (symmetric) kernel function by default adds the current status values between [-1,1].

The `kernel_unif_reflective` is similar to `kernel_unif` with the main difference that proposals are bounded to be within [lb, ub].

Value

An object of class fmcmc_kernel. `fmcmc_kernel` objects are intended to be used with the `MCMC()` function.

See Also

Other kernels: `kernel_adapt()`, `kernel_mirror`, `kernel_new()`, `kernel_normal()`, `kernel_ram()`

Examples

```
# Multivariate setting with 4 parameters in which we set the kernel to make
# proposals one parameter at-a-time in a random ordering.
kern <- kernel_unif(scheme = "random")
```

`MCMC` *Markov Chain Monte Carlo*

Description

A flexible implementation of the Metropolis-Hastings MCMC algorithm.

Usage

```
MCMC(
  initial,
  fun,
  nsteps,
  ...,
  nchains = 1L,
  burnin = 0L,
  thin = 1L,
  kernel = kernel_normal(),
  multicore = FALSE,
  conv_checker = NULL,
  cl = NULL,
  progress = interactive() && !multicore,
  chain_id = 1L
)

## S3 method for class 'mcmc'
MCMC(
  initial,
  fun,
  nsteps,
  ...,
  nchains = 1L,
  burnin = 0L,
  thin = 1L,
  kernel = kernel_normal(),
  multicore = FALSE,
  conv_checker = NULL,
  cl = NULL,
  progress = interactive() && !multicore,
  chain_id = 1L
)

## S3 method for class 'mcmc.list'
MCMC(
  initial,
  fun,
  nsteps,
  ...,
  nchains = 1L,
  burnin = 0L,
  thin = 1L,
  kernel = kernel_normal(),
  multicore = FALSE,
  conv_checker = NULL,
  cl = NULL,
  progress = interactive() && !multicore,
  chain_id = 1L
)

## Default S3 method:
MCMC(
  initial,
  fun,
  nsteps,
  ...,
  nchains = 1L,
  burnin = 0L,
  thin = 1L,
  kernel = kernel_normal(),
  multicore = FALSE,
  conv_checker = NULL,
  cl = NULL,
  progress = interactive() && !multicore,
  chain_id = 1L
)
```

Arguments

`initial`	Either a numeric matrix or vector, or an object of class coda::mcmc or coda::mcmc.list (see details). initial values of the parameters for each chain (See details).
`fun`	A function. Returns the log-likelihood.
`nsteps`	Integer scalar. Length of each chain.
`...`	Further arguments passed to `fun`.
`nchains`	Integer scalar. Number of chains to run (in parallel).
`burnin`	Integer scalar. Length of burn-in. Passed to coda::mcmc as `start`.
`thin`	Integer scalar. Passed to coda::mcmc.
`kernel`	An object of class fmcmc_kernel.
`multicore`	Logical. If `FALSE` then chains will be executed in serial.
`conv_checker`	A function that receives an object of class coda::mcmc.list, and returns a logical value with `TRUE` indicating convergence. See the "Automatic stop" section and the convergence-checker manual.
`cl`	A `cluster` object passed to parallel::clusterApply.
`progress`	Logical scalar. When set to `TRUE` shows a progress bar. A new bar will be show every time that the convergence checker is called.
`chain_id`	Integer scalar (internal use only). This is an argument passed to the kernel function and it allows it identify in which of the chains the process is taking place. This could be relevant for some kernels (see `kernel_new()`).

Details

This function implements MCMC using the Metropolis-Hastings ratio with flexible transition kernels. Users can specify either one of the available transition kernels or define one of their own (see kernels). Furthermore, it allows easy parallel implementation running multiple chains in parallel. In addition, we incorporate a variety of convergence diagnostics, alternatively the user can specify their own (see convergence-checker).

We now give details of the various options included in the function.

Starting point

By default, if `initial` is of class `mcmc`, `MCMC` will take the last `nchains` points from the chain as starting point for the new sequence. If `initial` is of class `mcmc.list`, the number of chains in `initial` must match the `nchains` parameter.

If `initial` is a vector, then it must be of length equal to the number of parameters used in the model. When using multiple chains, if `initial` is not an object of class `mcmc` or `mcmc.list`, then it must be a numeric matrix with as many rows as chains, and as many columns as parameters in the model.

Multiple chains

When `nchains` > 1, the function will run multiple chains. Furthermore, if `cl` is not passed, `MCMC` will create a `PSOCK` cluster using parallel::makePSOCKcluster with parallel::detectCores clusters and attempt to execute using multiple cores. Internally, the function does the following:

```
# Creating the cluster
ncores <- parallel::detectCores()
ncores <- ifelse(nchains < ncores, nchains, ncores)
cl     <- parallel::makePSOCKcluster(ncores)

# Loading the package and setting the seed using clusterRNGStream
invisible(parallel::clusterEvalQ(cl, library(fmcmc)))
parallel::clusterSetRNGStream(cl, .Random.seed)
```

When running in parallel, objects that are used within `fun` must be passed through `...`, otherwise the cluster will return with an error.

The user controls the initial value of the parameters of the MCMC algorithm using the argument `initial`. When using multiple chains, i.e., `nchains` > 1, the user can specify multiple starting points, which is recommended. In such a case, each row of `initial` is use as a starting point for each of the chains. If `initial` is a vector and `nchains` > 1, the value is recycled, so all chains start from the same point (not recommended, the function throws a warning message).

Automatic stop

By default, no automatic stop is implemented. If one of the functions in convergence-checker is used, then the MCMC is done by bulks as specified by the convergence checker function, and thus the algorithm will stop if, the `conv_checker` returns `TRUE`. For more information see convergence-checker.

Value

An object of class coda::mcmc from the **coda** package. The `mcmc` object is a matrix with one column per parameter, and `nsteps` rows. If `nchains` > 1, then it returns a coda::mcmc.list.

References

Brooks, S., Gelman, A., Jones, G. L., & Meng, X. L. (2011). Handbook of Markov Chain Monte Carlo. Handbook of Markov Chain Monte Carlo.

Examples

```
# Univariate distributed data with multiple parameters ----------------------
# Parameters
```

```
set.seed(1231)
n <- 1e3
pars <- c(mean = 2.6, sd = 3)

# Generating data and writing the log likelihood function
D <- rnorm(n, pars[1], pars[2])
fun <- function(x) {
  x <- log(dnorm(D, x[1], x[2]))
  sum(x)
}

# Calling MCMC, but first, loading the coda R package for
# diagnostics
library(coda)
ans <- MCMC(
  fun, initial = c(mu=1, sigma=1), nsteps = 2e3,
  kernel = kernel_normal_reflective(scale = .1, ub = 10, lb = 0)
  )

# Ploting the output
oldpar <- par(no.readonly = TRUE)
par(mfrow = c(1,2))
boxplot(as.matrix(ans),
        main = expression("Posterior distribution of"~mu~and~sigma),
        names =  expression(mu, sigma), horizontal = TRUE,
        col  = blues9[c(4,9)],
        sub = bquote(mu == .(pars[1])~", and"~sigma == .(pars[2]))
)
abline(v = pars, col  = blues9[c(4,9)], lwd = 2, lty = 2)

plot(apply(as.matrix(ans), 1, fun), type = "l",
     main = "LogLikelihood",
     ylab = expression(L("{"~mu,sigma~"}"~"|"~D))
)
par(oldpar)


# In this example we estimate the parameter for a dataset with ----------------
# With 5,000 draws from a MVN() with parameters M and S.

# Loading the required packages
library(mvtnorm)
library(coda)

# Parameters and data simulation
S <- cbind(c(.8, .2), c(.2, 1))
M <- c(0, 1)

set.seed(123)
D <- rmvnorm(5e3, mean = M, sigma = S)

# Function to pass to MCMC
fun <- function(pars) {
  # Putting the parameters in a sensible way
  m <- pars[1:2]
  s <- cbind( c(pars[3], pars[4]), c(pars[4], pars[5]) )

  # Computing the unnormalized log likelihood
  sum(log(dmvnorm(D, m, s)))
}

# Calling MCMC
ans <- MCMC(
  initial = c(mu0=5, mu1=5, s0=5, s01=0, s2=5),
  fun,
  kernel  = kernel_normal_reflective(
    lb    = c(-10, -10, .01, -5, .01),
    ub    = 5,
    scale = 0.01
  ),
  nsteps  = 1e4,
  thin    = 20,
  burnin  = 5e3
)

# Checking out the outcomes
plot(ans)
summary(ans)

# Multiple chains -----------------------------------------------------------

# As we want to run -fun- in multiple cores, we have to
# pass -D- explicitly (unless using Fork Clusters)
# just like specifying that we are calling a function from the
# -mvtnorm- package.

fun <- function(pars, D) {
  # Putting the parameters in a sensible way
  m <- pars[1:2]
  s <- cbind( c(pars[3], pars[4]), c(pars[4], pars[5]) )

  # Computing the unnormalized log likelihood
  sum(log(mvtnorm::dmvnorm(D, m, s)))
}

# Two chains
ans <- MCMC(
  initial = c(mu0=5, mu1=5, s0=5, s01=0, s2=5),
  fun,
  nchains = 2,
  kernel  = kernel_normal_reflective(
    lb    = c(-10, -10, .01, -5, .01),
    ub    = 5,
    scale = 0.01
  ),
  nsteps  = 1e4,
  thin    = 20,
  burnin  = 5e3,
  D       = D
)

summary(ans)
```

`new_progress_bar` *Progress bar*

Description

A simple progress bar. This function is used in MCMC when the function has been called with a single processor.

Usage

```
new_progress_bar(
  n,
  probs = c(0, 0.25, 0.5, 0.75, 1),
  width = getOption("width", 80),
  symbol = "/",
  ...
)
```

Arguments

`n`	Integer. Number of steps.
`probs`	Double vector. Quantiles where to put marks
`width`	Integer. Width of the bar in characters.
`symbol`	Character. Single character symbol to print each bar.
`...`	Further arguments passed to `cat()` such as `file` and append.

Value

A function that can be included at the interior of a loop to mark the progress of the loop. It receives a single argument, `i`, which is the number of the current step.

Examples

```
x <- new_progress_bar(20)
for (i in 1:20) {
  Sys.sleep(2/20)
  x(i)
}
```

`plan_update_sequence` *Parameters' update sequence*

Description

Parameters' update sequence

Usage

```
plan_update_sequence(k, nsteps, fixed, scheme)
```

Arguments

`k`	Integer. Number of parameters
`nsteps`	Integer. Number of steps.
`fixed`	Logical scalar or vector of length `k`. Indicates which parameters will be treated as fixed or not. Single values are recycled.
`scheme`	Scheme in which the proposals are made (see details).

Details

The parameter `scheme` present on the currently available kernels sets the way in which proposals are made. By default, `scheme = "joint"`, proposals are done jointly, this is, at each step of the chain we are proposing new states for each parameter of the model. When `scheme = "ordered"`, a sequential update schema is followed, in which, at each step of the chain, proposals are made one variable at a time, If `scheme = "random"`, proposals are also made one variable at a time but in a random scheme.

Finally, users can specify their own sequence of proposals for the variables by passing a numeric vector to `scheme`, for example, if the user wants to make sequential proposals following the scheme 2, 1, 3, then scheme must be set to be `scheme = c(2,1,3)`.

Value

A logical vector of size `nsteps` x `k`.

`reflect_on_boundaries` *Reflective Boundaries*

Description

Adjust a proposal according to its support by reflecting it. This is the workhorse of kernel_normal_reflective and kernel_unif_reflective. It is intended for internal use only.

Usage

```
reflect_on_boundaries(x, lb, ub, which)
```

Arguments

`x`	A numeric vector. The proposal
`lb, ub`	Numeric vectors of length `length(x)`. Lower and upper bounds.
`which`	Integer vector. Index of variables to be updated.

Value

An adjusted proposal vector.

Index