Package 'fmcmc'

August 27, 2019

Title A friendly MCMC framework

Version 0.2-0

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).

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, mvtnorm, knitr, rmarkdown, mcmc, tinytest

Imports parallel, coda, stats, methods

RoxygenNote 6.1.1

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 2019-08-27 08:50:05 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.

`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

---

kernels *Various kernel functions for MCMC*

Description

Various kernel functions for MCMC

Usage

```
kernel_new(proposal, logratio = NULL, ...)

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

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")

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

`proposal, logratio`	Functions. The function receives 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.
`x`	An object of class `fmcmc_kernel`.
`min., max.`	Passed to stats::runif.
`fixed`	Logical scalar or vector. When `TRUE` fixes the corresponding parameter, avoiding new proposals.
`scheme`	scheme in which proposals are made (see details).
`lb, ub`	Either a numeric vector or a scalar. Lower and upper bounds for bounded kernels.
`mu, scale`	Either a numeric vector or a scalar. Proposal mean and scale.

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.
- `logration`: 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, logratio 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 environemnt 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 more details see the vignette `vignette("user-defined-kernels","fmcmc")`.

Proposal scheme

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)`.

Creating your own kernels

The function `kernel_new` is a helper function that allows creating `fmcmc_kernel` which is used with the `MCMC` function. The `fmcmc_kernel` are the backbone of the MCMC function.

Kernels

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]`.

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).

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.
```

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())

## 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)

## 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)

## 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)
```

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 (currently ignored).

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.

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.

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.

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)
```

`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`	Numberic vectors of length `length(x)`. Lower and upper bounds.
`which`	Integer vector. Index of variables to be updated.

Value

An adjusted proposal vector.

Index