

# Package ‘adaptMCMC’

October 12, 2022

**Type** Package

**Title** Implementation of a Generic Adaptive Monte Carlo Markov Chain Sampler

**Version** 1.4

**Date** 2021-03-29

**Author** Andreas Scheidegger, <andreas.scheidegger@eawag.ch>, <scheidegger.a@gmail.com>

**Maintainer** Andreas Scheidegger <andreas.scheidegger@eawag.ch>

**Description** Enables sampling from arbitrary distributions if the log density is known up to a constant; a common situation in the context of Bayesian inference. The implemented sampling algorithm was proposed by Vihola (2012) <DOI:10.1007/s11222-011-9269-5> and achieves often a high efficiency by tuning the proposal distributions to a user defined acceptance rate.

**License** GPL (>= 2)

**LazyLoad** yes

**Depends** R (>= 2.14.1), parallel, coda, Matrix

**Imports** rancmc

**URL** <https://github.com/scheidan/adaptMCMC>

**BugReports** <https://github.com/scheidan/adaptMCMC/issues>

**RoxygenNote** 7.1.1

**NeedsCompilation** no

**Repository** CRAN

**Date/Publication** 2021-03-29 09:00:03 UTC

## R topics documented:

adaptMCMC-package . . . . .	2
convert.to.coda . . . . .	3
MCMC . . . . .	4
MCMC.add.samples . . . . .	7
MCMC.parallel . . . . .	8

<b>Index</b>	<b>11</b>
--------------	-----------

---

adaptMCMC-package      *Generic adaptive Monte Carlo Markov Chain sampler*

---

## Description

Enables sampling from arbitrary distributions if the log density is known up to a constant; a common situation in the context of Bayesian inference. The implemented sampling algorithm was proposed by Vihola (2012) and achieves often a high efficiency by tuning the proposal distributions to a user defined acceptance rate.

## Details

Package:	adaptMCMC
Type:	Package
Version:	1.4
Date:	2021-03-29
License:	GPL (>= 2)
LazyLoad:	yes

The workhorse function is `MCMC`. Chains can be updated with `MCMC.add.samples`. `MCMC.parallel` is a wrapper to generate independent chains on several CPU's in parallel using `parallel`. `coda`-functions can be used after conversion with `convert.to.coda`.

## Author(s)

Andreas Scheidegger, <andreas.scheidegger@eawag.ch> or <scheidegger.a@gmail.com>

## References

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

## See Also

`MCMC`, `MCMC.add.samples`, `MCMC.parallel`, `convert.to.coda`

The package `HI` provides an adaptive rejection Metropolis sampler with the function `arms`. See also `Metro_Hastings` of the `MHadaptive` package.

---

convert.to.coda	<i>Converts chain(s) into <b>coda</b> objects.</i>
-----------------	--

---

## Description

Converts chain(s) produced by MCMC or MCMC.parallel into **coda** objects.

## Usage

```
convert.to.coda(sample)
```

## Arguments

sample            output of MCMC or MCMC.parallel.

## Details

Converts chain(s) produced by MCMC or MCMC.parallel so that they can be used with functions of the **coda** package.

## Value

An object of the class mcmc or mcmc.list.

## Author(s)

Andreas Scheidegger, <andreas.scheidegger@eawag.ch> or <scheidegger.a@gmail.com>

## See Also

[MCMC](#), [mcmc](#), [mcmc.list](#)

## Examples

```
## -----
## Banana shaped distribution

## log-pdf to sample from
p.log <- function(x) {
  B <- 0.03 # controls 'bananacity'
  -x[1]^2/200 - 1/2*(x[2]+B*x[1]^2-100*B)^2
}

## -----
## generate 200 samples

samp <- MCMC(p.log, n=200, init=c(0, 1), scale=c(1, 0.1),
            adapt=TRUE, acc.rate=0.234)
```

```
## -----
## convert in object of class 'mcmc'
samp.coda <- convert.to.coda(samp)

class(samp.coda)

## -----
## use functions of package 'coda'

require(coda)

plot(samp.coda)
cumuplot(samp.coda)
```

---

MCMC

*(Adaptive) Metropolis Sampler*


---

## Description

Implementation of the robust adaptive Metropolis sampler of Vihola (2012).

## Usage

```
MCMC(p, n, init, scale = rep(1, length(init)),
     adapt = !is.null(acc.rate), acc.rate = NULL, gamma = 2/3,
     list = TRUE, showProgressBar=interactive(), n.start = 0, ...)
```

## Arguments

<code>p</code>	function that returns a value proportional to the log probability density to sample from. Alternatively it can be a function that returns a list with at least one element named <code>log.density</code> . See details below.
<code>n</code>	number of samples.
<code>init</code>	vector with initial values.
<code>scale</code>	vector with the variances <i>or</i> covariance matrix of the jump distribution.
<code>adapt</code>	if TRUE, adaptive sampling is used, if FALSE classic metropolis sampling, if a positive integer the adaption stops after <code>adapt</code> iterations.
<code>acc.rate</code>	desired acceptance rate (ignored if <code>adapt=FALSE</code> )
<code>gamma</code>	controls the speed of adaption. Should be between 0.5 and 1. A lower gamma leads to faster adaption.
<code>list</code>	logical. If TRUE a list is returned otherwise only a matrix with the samples.
<code>showProgressBar</code>	logical. If TRUE a progress bar is shown.
<code>n.start</code>	iteration where the adaption starts. Only internally used.
<code>...</code>	further arguments passed to <code>p</code> .

## Details

The algorithm tunes the covariance matrix of the (normal) jump distribution to achieve the desired acceptance rate. Classic (non-adaptive) Metropolis sampling can be obtained by setting `adapt=FALSE`.

Note, due to the calculation for the adaption steps the sampler is rather slow. However, with a suitable jump distribution good mixing can be observed with less samples. This is crucial if the computation of  $p$  is slow.

In some cases the function  $p$  may not only calculate the log density but return a list containing also other values. For example if  $p$  is a log posterior one may be also interested to store the corresponding prior and likelihood values. The function must either return always a scalar or always a list, however, the length of the list may vary.

## Value

If `list=FALSE` a matrix is with the samples.

If `list=TRUE` a list is returned with the following components:

<code>samples</code>	matrix with samples
<code>log.p</code>	vector with the (unnormalized) log density for each sample
<code>n.sample</code>	number of generated samples
<code>acceptance.rate</code>	acceptance rate
<code>adaption</code>	either logical if adaption was used or not, or the number of adaption steps.
<code>sampling.parameters</code>	a list with further sampling parameters. Mainly used by <code>MCMC.add.samples()</code>
.	.
<code>extra.values</code>	A list containing additional return values provided by $p$ . Only if $p$ provides a list.

## Note

Due to numerical errors it may happen that the computed covariance matrix is not positive definite. In such a case the nearest positive definite matrix is calculated with `nearPD()` from the package **Matrix**.

## Author(s)

Andreas Scheidegger, <[andreas.scheidegger@eawag.ch](mailto:andreas.scheidegger@eawag.ch)> or <[scheidegger.a@gmail.com](mailto:scheidegger.a@gmail.com)>.

Thanks to David Pleydell, Venelin, and Umberto Picchini for spotting errors and providing improvements. Ian Taylor implemented the usage of `adapt_S` which lead to a nice speedup.

## References

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

**See Also**

[MCMC.parallel](#), [MCMC.add.samples](#)

The package HI provides an adaptive rejection Metropolis sampler with the function [arms](#). See also [Metro\\_Hastings](#) of the MHadaptive package.

**Examples**

```
## -----
## Banana shaped distribution

## log-pdf to sample from
p.log <- function(x) {
  B <- 0.03 # controls 'bananacity'
  -x[1]^2/200 - 1/2*(x[2]+B*x[1]^2-100*B)^2
}

## -----
## generate samples

## 1) non-adaptive sampling
samp.1 <- MCMC(p.log, n=200, init=c(0, 1), scale=c(1, 0.1),
              adapt=FALSE)

## 2) adaptive sampling
samp.2 <- MCMC(p.log, n=200, init=c(0, 1), scale=c(1, 0.1),
              adapt=TRUE, acc.rate=0.234)

## -----
## summarize results

str(samp.2)
summary(samp.2$samples)

## covariance of last jump distribution
samp.2$cov.jump

## -----
## plot density and samples

x1 <- seq(-15, 15, length=80)
x2 <- seq(-15, 15, length=80)
d.banana <- matrix(apply(expand.grid(x1, x2), 1, p.log), nrow=80)

par(mfrow=c(1,2))
image(x1, x2, exp(d.banana), col=cm.colors(60), asp=1, main="no adaption")
contour(x1, x2, exp(d.banana), add=TRUE, col=gray(0.6))
lines(samp.1$samples, type='b', pch=3)

image(x1, x2, exp(d.banana), col=cm.colors(60), asp=1, main="with adaption")
```

```

contour(x1, x2, exp(d.banana), add=TRUE, col=gray(0.6))
lines(samp.2$samples, type='b', pch=3)

## -----
## function returning extra information in a list

p.log.list <- function(x) {
  B <- 0.03 # controls 'bananacity'
  log.density <- -x[1]^2/200 - 1/2*(x[2]+B*x[1]^2-100*B)^2

  result <- list(log.density=log.density)

  ## under some conditions one may want to return other infos
  if(x[1]<0) {
    result$message <- "Attention x[1] is negative!"
    result$x <- x[1]
  }

  result
}

samp.list <- MCMC(p.log.list, n=200, init=c(0, 1), scale=c(1, 0.1),
  adapt=TRUE, acc.rate=0.234)

## the additional values are stored under `extras.values`
head(samp.list$extras.values)

```

---

MCMC.add.samples	<i>Add samples to an existing chain.</i>
------------------	--

---

### Description

Add samples to an existing chain produced by MCMC or MCMC.parallel.

### Usage

```
MCMC.add.samples(MCMC.object, n.update, ...)
```

### Arguments

MCMC.object	a list produced by MCMC or MCMC.parallel with option list = TRUE.
n.update	number of additional samples.
...	further arguments passed to p.

**Details**

Only objects generated with the option `list = TRUE` can be updated.

A list of chains produced by `MCMC.parallel` can be updated. However, the calculations are *not* performed in parallel (i.e. only a single CPU is used).

**Value**

A updated version of `MCMC`.object.

**Author(s)**

Andreas Scheidegger, <andreas.scheidegger@eawag.ch> or <scheidegger.a@gmail.com>

**See Also**

[MCMC](#), [MCMC.parallel](#)

**Examples**

```
## -----
## Banana shaped distribution

## log-pdf to sample from
p.log <- function(x) {
  B <- 0.03 # controls 'bananacity'
  -x[1]^2/200 - 1/2*(x[2]+B*x[1]^2-100*B)^2
}

## -----
## generate 200 samples

samp <- MCMC(p.log, n=200, init=c(0, 1), scale=c(1, 0.1),
             adapt=TRUE, acc.rate=0.234, list=TRUE)

## -----
## add 200 to the existing chain
samp <- MCMC.add.samples(samp, n.update=200)

str(samp)
```

---

MCMC.parallel

*Parallel computation of MCMC()*

---

**Description**

A wrapper function to generate several independent Markov chains by setting up cluster on a multi-core machine. The function is based on the **parallel** package.



**Usage**

```
MCMC.parallel(p, n, init, n.chain = 4, n.cpu, packages = NULL, dyn.libs=NULL,
  scale = rep(1, length(init)), adapt = !is.null(acc.rate),
  acc.rate = NULL, gamma = 2/3, list = TRUE, ...)
```

**Arguments**

<code>p</code>	function that returns a value proportional to the log probability density to sample from. Alternatively the function can return a list with at least one element named <code>log.density</code> .
<code>n</code>	number of samples.
<code>init</code>	vector with initial values.
<code>n.chain</code>	number of independent chains.
<code>n.cpu</code>	number of CPUs that should be used in parallel.
<code>packages</code>	vector with name of packages to load into each instance. (Typically, all packages on which <code>p</code> depends.)
<code>dyn.libs</code>	vector with name of dynamic link libraries (shared objects) to load into each instance. The libraries must be located in the working directory.
<code>scale</code>	vector with the variances <i>or</i> covariance matrix of the jump distribution.
<code>adapt</code>	if TRUE, adaptive sampling is used, if FALSE classic metropolis sampling, if a positive integer the adaption stops after <code>adapt</code> iterations.
<code>acc.rate</code>	desired acceptance rate (ignored if <code>adapt=FALSE</code> )
<code>gamma</code>	controls the speed of adaption. Should be between 0.5 and 1. A lower gamma leads to faster adaption.
<code>list</code>	logical. If TRUE a list of lists is returned otherwise a list of matrices with the samples.
<code>...</code>	further arguments passed to <code>p</code>

**Details**

This function is just a wrapper to use MCMC in parallel. It is based on **parallel**. Obviously, the application of this function makes only sense on a multi-core machine.

**Value**

A list with a list or matrix for each chain. See [MCMC](#) for details.

**Author(s)**

Andreas Scheidegger, <andreas.scheidegger@eawag.ch> or <scheidegger.a@gmail.com>

**See Also**

[MCMC](#)

**Examples**

```
## -----  
## Banana shaped distribution  
  
## log-pdf to sample from  
p.log <- function(x) {  
  B <- 0.03 # controls 'bananacity'  
  -x[1]^2/200 - 1/2*(x[2]+B*x[1]^2-100*B)^2  
}  
  
## -----  
## generate samples  
## compute 4 independent chains on 2 CPU's (if available) in parallel  
  
samp <- MCMC.parallel(p.log, n=200, init=c(x1=0, x2=1),  
  n.chain=4, n.cpu=2, scale=c(1, 0.1),  
  adapt=TRUE, acc.rate=0.234)  
  
str(samp)
```

# Index

`adapt_S`, [5](#)  
`adaptMCMC` (`adaptMCMC`-package), [2](#)  
`adaptMCMC`-package, [2](#)  
`arms`, [2, 6](#)

`convert.to.coda`, [2, 3](#)

`MCMC`, [2, 3, 4, 8, 9](#)  
`mcmc`, [3](#)  
`MCMC.add.samples`, [2, 6, 7](#)  
`mcmc.list`, [3](#)  
`MCMC.parallel`, [2, 6, 8, 8](#)  
`Metro_Hastings`, [2, 6](#)