Package ‘mcmc’

March 10, 2019

Version 0.9-6
Date 2019-03-09
Title Markov Chain Monte Carlo
Author Charles J. Geyer <charlie@stat.umn.edu> and Leif T. Johnson
<ltjohnson@google.com>
Maintainer Charles J. Geyer <charlie@stat.umn.edu>
Depends R (>= 3.0.2)
Imports stats
Suggests xtable, Iso
ByteCompile TRUE
Description Simulates continuous distributions of random vectors using
Markov chain Monte Carlo (MCMC). Users specify the distribution by an
R function that evaluates the log unnormalized density. Algorithms
are random walk Metropolis algorithm (function metrop), simulated
tempering (function temper), and morphometric random walk Metropolis
(Johnson and Geyer, 2012, <doi:10.1214/12-AOS1048>),
function morph.metrop),
which achieves geometric ergodicity by change of variable.
License MIT + file LICENSE
URL http://www.stat.umn.edu/geyer/mcmc/,
https://github.com/cjgeyer/mcmc
NeedsCompilation yes
Repository CRAN
Date/Publication 2019-03-10 07:20:03 UTC

R topics documented:

foo .............................................................. 2
initseq .......................................................... 2
logit ............................................................. 4
Simulated logistic regression data.

Description
Like it says

Usage
data(foo)

Format
A data frame with variables
- x1 quantitative predictor.
- x2 quantitative predictor.
- x3 quantitative predictor.
- y Bernoulli response.

Examples
library(mcmc)
data(foo)
out <- glm(y ~ x1 + x2 + x3, family = binomial, data = foo)
summary(out)

Initial Sequence Estimators

Description

Usage
initseq(x)
Arguments

\(x\) a numeric vector that is a scalar-valued functional of a reversible Markov chain.

Details

Let

\[ \gamma_k = \text{cov}(X_i, X_{i+k}) \]

considered as a function of the lag \(k\) be the autocovariance function of the input time series. Define

\[ \Gamma_k = \gamma_{2k} + \gamma_{2k+1} \]

the sum of consecutive pairs of autocovariances. Then Theorem 3.1 in Geyer (1992) says that \(\Gamma_k\) considered as a function of \(k\) is strictly positive, strictly decreasing, and strictly convex, assuming the input time series is a scalar-valued functional of a reversible Markov chain. All of the MCMC done by this package is reversible. This R function estimates the “big gamma” function, \(\Gamma_k\) considered as a function of \(k\), subject to three different constraints, (1) nonnegative, (2) nonnegative and nonincreasing, and (3) nonnegative, nonincreasing, and convex. It also estimates the variance in the Markov chain central limit theorem (CLT)

\[ \gamma_0 + 2 \sum_{k=1}^{\infty} \gamma_k = -\gamma_0 + 2 \sum_{k=0}^{\infty} \Gamma_k \]

Note: The batch means provided by \texttt{metrop} are also scalar functionals of a reversible Markov chain. Thus these initial sequence estimators applied to the batch means give valid standard errors for the mean of the batch means even when the batch length is too short to provide a valid estimate of asymptotic variance. One does, of course, have to multiply the asymptotic variance of the batch means by the batch length to get the asymptotic variance for the unbatched chain.

Value

a list containing the following components:

- \(\text{gamma0}\) the scalar \(\gamma_0\), the marginal variance of \(x\).
- \(\text{Gamma.pos}\) the vector \(\Gamma\), estimated so as to be nonnegative, where, as always, R uses one-origin indexing so \text{Gamma,pos}[1] is \(\Gamma_0\).
- \(\text{Gamma.dec}\) the vector \(\Gamma\), estimated so as to be nonnegative and nonincreasing, where, as always, R uses one-origin indexing so \text{Gamma.dec}[1] is \(\Gamma_0\).
- \(\text{Gamma.con}\) the vector \(\Gamma\), estimated so as to be nonnegative and nonincreasing and convex, where, as always, R uses one-origin indexing so \text{Gamma.con}[1] is \(\Gamma_0\).
- \(\text{var.pos}\) the scalar \(\text{gamma0 + 2 * sum(Gamma.pos)}\), which is the asymptotic variance in the Markov chain CLT. Divide by \(\text{length(x)}\) to get the approximate variance of the sample mean of \(x\).
- \(\text{var.dec}\) the scalar \(\text{gamma0 + 2 * sum(Gamma.dec)}\), which is the asymptotic variance in the Markov chain CLT. Divide by \(\text{length(x)}\) to get the approximate variance of the sample mean of \(x\).
- \(\text{var.con}\) the scalar \(\text{gamma0 + 2 * sum(Gamma.con)}\), which is the asymptotic variance in the Markov chain CLT. Divide by \(\text{length(x)}\) to get the approximate variance of the sample mean of \(x\).


**Bugs**

Not precisely a bug, but var.pos, var.dec, and var.con can be negative. This happens only when the chain is way too short to estimate the variance, and even then rarely. But it does happen.

**References**


**See Also**

metrop

**Examples**

```r
n <- 2e4
rho <- 0.99
x <- arima.sim(model = list(ar = rho), n = n)
out <- initseq(x)
## Not run:
plot(seq(along = out$Gamma.pos) - 1, out$Gamma.pos, 
     xlab = "k", ylab = expression(Gamma[k]), type = "l")
lines(seq(along = out$Gamma.dec) - 1, out$Gamma.dec, col = "red")
lines(seq(along = out$Gamma.con) - 1, out$Gamma.con, col = "blue")
## End(Not run)
# asymptotic 95% confidence interval for mean of x
mean(x) + c(-1, 1) * qnorm(0.975) * sqrt(out$var.con / length(x))
# estimated asymptotic variance
out$var.con
# theoretical asymptotic variance
(1 + rho) / (1 - rho) * 1 / (1 - rho^2)
# illustrating use with batch means
bm <- apply(matrix(x, nrow = 5), 2, mean)
initseq(bm)$var.con * 5
```

---

**logit**

*Simulated logistic regression data.*

**Description**

Like it says

**Usage**

data(logit)
**Format**

A data frame with variables
- **x1** quantitative predictor.
- **x2** quantitative predictor.
- **x3** quantitative predictor.
- **x4** quantitative predictor.
- **y** Bernoulli response.

**Examples**

```r
library(mcmc)
data(logit)
out <- glm(y ~ x1 + x2 + x3 + x4, family = binomial, data = logit)
summary(out)
```

---

**Description**

Markov chain Monte Carlo for continuous random vector using a Metropolis algorithm.

**Usage**

```r
metrop(obj, initial, nbatch, blen = 1, nspac = 1, scale = 1, outfun, debug = FALSE, ...)
```

### S3 method for class 'function'

```r
metrop(obj, initial, nbatch, blen = 1, nspac = 1, scale = 1, outfun, debug = FALSE, ...)
```

### S3 method for class 'metropolis'

```r
metrop(obj, initial, nbatch, blen = 1, nspac = 1, scale = 1, outfun, debug = FALSE, ...)
```

**Arguments**

- **obj** Either an R function or an object of class "metropolis" from a previous invocation of this function. If a function, it evaluates the log unnormalized probability density of the desired equilibrium distribution of the Markov chain. Its first argument is the state vector of the Markov chain. Other arguments arbitrary and taken from the ... arguments of this function. It should return -Inf for points of the state space having probability zero under the desired equilibrium distribution. See also Details and Warning.

If an object of class "metropolis", any missing arguments (including the log unnormalized density function) are taken from this object. Also initial is ignored and the initial state of the Markov chain is the final state from the run recorded in obj.
initial  a real vector, the initial state of the Markov chain. Must be feasible, see Details. Ignored if obj is of class "metropolis".
nbatch  the number of batches.
blen  the length of batches.
nspac  the spacing of iterations that contribute to batches.
scale  controls the proposal step size. If scalar or vector, the proposal is $x + scale \ast z$ where $x$ is the current state and $z$ is a standard normal random vector. If matrix, the proposal is $x \ast scale \ast z$.
outfun  controls the output. If a function, then the batch means of outfun(state, ...) are returned. If a numeric or logical vector, then the batch means of state[outfun] (if this makes sense). If missing, the the batch means of state are returned.
debug  if TRUE extra output useful for testing.
...  additional arguments for obj or outfun.

Details

Runs a “random-walk” Metropolis algorithm, terminology introduced by Tierney (1994), with multivariate normal proposal producing a Markov chain with equilibrium distribution having a specified unnormalized density. Distribution must be continuous. Support of the distribution is the support of the density specified by argument obj. The initial state must satisfy obj(state)^Minf. Description of a complete MCMC analysis (Bayesian logistic regression) using this function can be found in the vignette vignette("demo", "mcmc").

Suppose the function coded by the log unnormalized function (either obj or obj$llud) is actually a log unnormalized density, that is, if $w$ denotes that function, then $e^w$ integrates to some value strictly between zero and infinity. Then the metrop function always simulates a reversible, Harris ergodic Markov chain having the equilibrium distribution with this log unnormalized density. The chain is not guaranteed to be geometrically ergodic. In fact it cannot be geometrically ergodic if the tails of the log unnormalized density are sufficiently heavy. The morph.metrop function deals with this situation.

Value

an object of class "mcmc", subclass "metropolis", which is a list containing at least the following components:

accept  fraction of Metropolis proposals accepted.
batch  nbatch by p matrix, the batch means, where p is the dimension of the result of outfun if outfun is a function, otherwise the dimension of state[outfun] if that makes sense, and the dimension of state when outfun is missing.

accept.batch  a vector of length nbatch, the batch means of the acceptances.
initial  value of argument initial.
fine  final state of Markov chain.
initial.seed  value of .Random.seed before the run.
final.seed  value of .Random.seed after the run.
time  running time of Markov chain from system.time().
the function used to calculate log unnormalized density, either \texttt{obj} or \texttt{obj$\text{lad}} from a previous run.

\begin{itemize}
  \item \texttt{nbatch} the argument \texttt{nbatch} or \texttt{obj$nbatch}.
  \item \texttt{blen} the argument \texttt{blen} or \texttt{obj$blen}.
  \item \texttt{nspac} the argument \texttt{nspac} or \texttt{obj$nspac}.
  \item \texttt{outfun} the argument \texttt{outfun} or \texttt{obj$outfun}.
\end{itemize}

Description of additional output when \texttt{debug = TRUE} can be found in the vignette \texttt{debug (.doc/ debug.pdf)}.

\section*{Warning}

If \texttt{outfun} is missing or not a function, then the log unnormalized density can be defined without a ...argument and that works fine. One can define it starting \texttt{ludfun <- function(state)} and that works or \texttt{ludfun <- function(state, foo, bar)}, where \texttt{foo} and \texttt{bar} are supplied as additional arguments to \texttt{metrop}.

If \texttt{outfun} is a function, then both it and the log unnormalized density function can be defined without ...arguments if they have exactly the same arguments list and that works fine. Otherwise it doesn’t work. Define these functions by

\begin{verbatim}
ludfun <- function(state, foo)
outfun <- function(state, bar)
\end{verbatim}

and you get an error about unused arguments. Instead define these functions by

\begin{verbatim}
ludfun <- function(state, foo, ...)
outfun <- function(state, bar, ...)
\end{verbatim}

and supply \texttt{foo} and \texttt{bar} as additional arguments to \texttt{metrop}, and that works fine.

In short, the log unnormalized density function and \texttt{outfun} need to have ...in their arguments list to be safe. Sometimes it works when ...is left out and sometimes it doesn’t.

Of course, one can avoid this whole issue by always defining the log unnormalized density function and \texttt{outfun} to have only one argument \texttt{state} and use global variables (objects in the \texttt{R} global environment) to specify any other information these functions need to use. That too follows the \texttt{R} way. But some people consider that bad programming practice.

A third option is to define either or both of these functions using a function factory. This is demonstrated in the vignette for this package named \texttt{demo}, which is shown by \texttt{vignette(“demo”, “mcmc”).}

\section*{Philosophy of MCMC}

This function follows the philosophy of MCMC explained the introductory chapter of the \textit{Handbook of Markov Chain Monte Carlo} (Geyer, 2011).

This function automatically does batch means in order to reduce the size of output and to enable easy calculation of Monte Carlo standard errors (MCSE), which measure error due to the Monte Carlo sampling (not error due to statistical sampling — MCSE gets smaller when you run the computer longer, but statistical sampling variability only gets smaller when you get a larger data set). All of
This function does not apparently do “burn-in” because this concept does not actually help with MCMC (Geyer, 2011, Section 1.11.4) but the re-entrant property of this function does allow one to do “burn-in” if one wants. Assuming `ludfun`, `start.value`, `scale` have been already defined and are, respectively, an R function coding the log unnormalized density of the target distribution, a valid state of the Markov chain, and a useful scale factor,

```r
out <- metrop(ludfun, start.value, nbatch = 1, blen = 1e5, scale = scale)
out <- metrop(out, nbatch = 100, blen = 1000)
```

throws away a run of 100 thousand iterations before doing another run of 100 thousand iterations that is actually useful for analysis, for example,

```r
apply(out$batch, 2, mean)
apply(out$batch, 2, sd) / sqrt(out$nbatch)
```

give estimates of posterior means and their MCSE assuming the batch length (here 1000) was long enough to contain almost all of the significant autocorrelation (see Geyer, 2011, Section 1.10, for more on MCSE). The re-entrant property of this function (the second run starts where the first one stops) assures that this is really “burn-in”.

The re-entrant property allows one to do very long runs without having to do them in one invocation of this function.

```r
out2 <- metrop(out)
out3 <- metrop(out2)
batch <- rbind(out$batch, out2$batch, out3$batch)
```

produces a result as if the first run had been three times as long.

### Tuning

The `scale` argument must be adjusted so that the acceptance rate is not too low or too high to get reasonable performance. The rule of thumb is that the acceptance rate should be about 25%. But this recommendation (Gelman, et al., 1996) is justified by analysis of a toy problem (simulating a spherical multivariate normal distribution) for which MCMC is unnecessary. There is no reason to believe this is optimal for all problems (if it were optimal, a stronger theorem could be proved). Nevertheless, it is clear that at very low acceptance rates the chain makes little progress (because in most iterations it does not move) and that at very high acceptance rates the chain also makes little progress (because unless the log unnormalized density is nearly constant, very high acceptance rates can only be achieved by very small values of `scale` so the steps the chain takes are also very small).

Even in the Gelman, et al. (1996) result, the optimal rate for spherical multivariate normal depends on dimension. It is 44% for \( d = 1 \) and 23% for \( d = \infty \). Geyer and Thompson (1995) have an example, admittedly for simulated tempering (see `temper`) rather than random-walk Metropolis, in which no acceptance rate less than 70% produces an ergodic Markov chain. Thus 25% is merely a rule of thumb. We only know we don’t want too high or too low. Probably 1% or 99% is very inefficient.
References


See Also

`morph.metrop` and `temper`

Examples

```r
h <- function(x) if (all(x >= 0) & sum(x) <= 1) return(1) else return(-Inf)
out <- metrop(h, rep(0, 5), 1000)
out$accept
# acceptance rate too low
out <- metrop(out, scale = 0.1)
out$accept
t.test(out$accept$batch$con$conf.int
# acceptance rate o. k. (about 25 percent)
plot(out$batch[, 1])
# but run length too short (few excursions from end to end of range)
out <- metrop(out, nbatch = 1e4)
out$accept
plot(out$batch[, 1])
hist(out$batch[, 1])
acf(out$batch[, 1], lag.max = 250)
# looks like batch length of 250 is perhaps OK
out <- metrop(out, blen = 250, nbatch = 100)
apply(out$batch, 2, mean) # Monte Carlo estimates of means
apply(out$batch, 2, sd) / sqrt(out$nbatch) # Monte Carlo standard errors
t.test(out$accept$batch$con$conf.int
acf(out$batch[, 1]) # appears that blen is long enough
```
Usage

morph(b, r, p, center)
morph.identity()

Arguments

b  Positive real number. May be missing.

r  Non-negative real number. May be missing. If p is specified, r defaults to 0.

p  Real number strictly greater than 2. May be missing. If r is specified, p defaults to 3.

center  Real scalar or vector. May be missing. If center is a vector it should be the same length of the state of the Markov chain, center defaults to 0

Details

The morph function facilitates using variable transformations by providing functions to (using \(X\) for the original random variable with the pdf \(f_X\), and \(Y\) for the transformed random variable with the pdf \(f_Y\)):

- Calculate the log unnormalized probability density for \(Y\) induced by the transformation.
- Transform an arbitrary function of \(X\) to a function of \(Y\).
- Transform values of \(X\) to values of \(Y\).
- Transform values of \(Y\) to values of \(X\) (the inverse transformation).

for a select few transformations.

morph.identity implements the identity transformation, \(Y = X\).

The parameters \(r\), \(p\), \(b\) and \(center\) specify the transformation function. In all cases, \(center\) gives the center of the transformation, which is the value \(c\) in the equation

\[ Y = f(X - c). \]

If no parameters are specified, the identity transformation, \(Y = X\), is used.

The parameters \(r\), \(p\) and \(b\) specify a function \(g\), which is a monotonically increasing bijection from the non-negative reals to the non-negative reals. Then

\[ f(X) = g(|X|) \frac{X}{|X|} \]

where \(|X|\) represents the Euclidean norm of the vector \(X\). The inverse function is given by

\[ f^{-1}(Y) = g^{-1}(|Y|) \frac{Y}{|Y|}. \]

The parameters \(r\) and \(p\) are used to define the function

\[ g_1(x) = x + (x - r)^p I(x > r) \]
where \( I(\cdot) \) is the indicator function. We require that \( r \) is non-negative and \( p \) is strictly greater than 2. The parameter \( b \) is used to define the function

\[
g_2(x) = (e^{bx} - e/3)I(x > \frac{1}{b}) + (x^3b^3e/6 + xbe/2)I(x \leq \frac{1}{b})
\]

We require that \( b \) is positive.

The parameters \( r, p \) and \( b \) specify \( f^{-1} \) in the following manner:

- If one or both of \( r \) and \( p \) is specified, and \( b \) is not specified, then
  \[
f^{-1}(X) = g_1(|X|) \frac{X}{|X|}.
\]
  If only \( r \) is specified, \( p = 3 \) is used. If only \( p \) is specified, \( r = 0 \) is used.

- If only \( b \) is specified, then
  \[
f^{-1}(X) = g_2(|X|) \frac{X}{|X|}.
\]

- If one or both of \( r \) and \( p \) is specified, and \( b \) is also specified, then
  \[
f^{-1}(X) = g_2(g_1(|X|)) \frac{X}{|X|}.
\]

**Value**

A list containing the functions

- `outfun(f)`, a function that operates on functions. `outfun(f)` returns the function `f(inverse(state), ...)`.
- `inverse`, the inverse transformation function.
- `transform`, the transformation function.
- `lud`, a function that operates on functions. As input, `lud` takes a function that calculates a log unnormalized probability density, and returns a function that calculates the log unnormalized density by transforming a random variable using the `transform` function. \( \text{lud}(f) = \text{function}(\text{state}, ...) \ f(\text{inverse}(\text{state}), ...) + \log.\text{jacobian}(\text{state}) \) where \( \log.\text{jacobian} \) represents the function that calculate the log Jacobian of the transformation. \( \log.\text{jacobian} \) is not returned.

**Warning**

The equations for the returned `transform` function (see below) do not have a general analytical solution when \( p \) is not equal to 3. This implementation uses numerical approximation to calculate `transform` when \( p \) is not equal to 3. If computation speed is a factor, it is advisable to use \( p=3 \). This is not a factor when using `morph.metrop`, as `transform` is only called once during setup, and not at all while running the Markov chain.

**See Also**

`morph.metrop`
Examples

# use an exponential transformation, centered at 100.
bl <- morph(b=1, center=100)
# original log unnormalized density is from a t distribution with 3
# degrees of freedom, centered at 100.
lud.transformed <- bl$lad(function(x) dt(x - 100, df=3, log=TRUE))
d.transformed <- Vectorize(function(x) exp(lud.transformed(x)))
## Not run:
curve(d.transformed, from=-3, to=3, ylab="Induced Density")

## End(Not run)

morph.metrop Morphometric Metropolis Algorithm

Description

Markov chain Monte Carlo for continuous random vector using a Metropolis algorithm for an induced density.

Usage

morph.metrop(obj, initial, nbatch, blen = 1, nspac = 1, scale = 1, outfun, debug = FALSE, morph, ...)

Arguments

obj see metrop.
initial see metrop.
nbatch see metrop.
blen see metrop.
nspac see metrop.
scale see metrop.
outfun unlike for metrop must be a function or missing; if missing the identity function, function(x) x, is used.
ddebug see metrop.
morph morph object used for transformations. See morph.
... see metrop.
morph.metrop implements morphometric methods for Markov chains. The caller specifies a log unnormalized probability density and a transformation. The transformation specified by the morph parameter is used to induce a new log unnormalized probability density, a Metropolis algorithm is run for the induced density. The Markov chain is transformed back to the original scale. Running the Metropolis algorithm for the induced density, instead of the original density, can result in a Markov chain with better convergence properties. For more details see Johnson and Geyer (submitted). Except for morph, all parameters are passed to metrop, transformed when necessary. The scale parameter is not transformed.

If \( X \) is a real vector valued continuous random variable, and \( Y = f(X) \) where \( f \) is a diffeomorphism, then the pdf of \( Y \) is given by

\[
f_Y(y) = f_X(f^{-1}(y))|\nabla f^{-1}(y)|
\]

where \( f_X \) is the pdf of \( X \) and \( \nabla f^{-1} \) is the Jacobian of \( f^{-1} \). Because \( f \) is a diffeomorphism, a Markov chain for \( f_Y \) may be transformed into a Markov chain for \( f_X \). Furthermore, these Markov chains are isomorphic (Johnson and Geyer, submitted) and have the same convergence rate. The morph variable provides a diffeomorphism, morph.metrop uses this diffeomorphism to induce the log unnormalized density, \( \log f_Y \) based on the user supplied log unnormalized density, \( \log f_X \). morph.metrop runs a Metropolis algorithm for \( \log f_Y \) and transforms the resulting Markov chain into a Markov chain for \( f_X \). The user accessible output components are the same as those that come from metrop, see the documentation for metrop for details.

Subsequent calls of morph.metrop may change to the transformation by specifying a new value for morph.

Any of the other parameters to morph.metrop may also be modified in subsequent calls. See metrop for more details.

The general idea is that a random-walk Metropolis sampler (what metrop does) will not be geometrically ergodic unless the tails of the unnormalized density decrease superexponentially fast (so the tails of the log unnormalized density decrease faster than linearly). It may not be geometrically ergodic even then (see Johnson and Geyer, submitted, for the complete theory). The transformations used by this function (provided by morph) can produce geometrically ergodic chains when the tails of the log unnormalized density are too light for metrop to do so.

When the tails of the unnormalized density are exponentially light but not superexponentially light (so the tails of the log unnormalized density are asymptotically linear, as in the case of exponential family models when conjugate priors are used, for example logistic regression, Poisson regression with log link, or log-linear models for categorical data), one should use morph with \( b = 0 \) (the default), which produces a transformation of the form \( g_1 \) in the notation used in the details section of the help for morph. This will produce a geometrically ergodic sampler if other features of the log unnormalized density are well behaved. For example it will do so for the exponential family examples mentioned above. (See Johnson and Geyer, submitted, for the complete theory.)

The transformation \( g_1 \) behaves like a shift transformation on a ball of radius \( r \) centered at center, so these arguments to morph should be chosen so that a sizable proportion of the probability under the original (untransformed) unnormalized density is contained in this ball. This function will work when \( r = 0 \) and center = 0 (the defaults) are used, but may not work as well as when \( r \) and center are well chosen.

When the tails of the unnormalized density are not exponentially light (so the tails of the log unnormalized density decrease sublinearly, as in the case of univariate and multivariate t distributions),
one should use `morph` with \( r > 0 \) and \( p = 3 \), which produces a transformation of the form \( g_2 \) composed with \( g_1 \) in the notation used in the details section of the help for `morph`. This will produce a geometrically ergodic sampler if other features of the log unnormalized density are well behaved. For example it will do so for the \( t \) examples mentioned above. (See Johnson and Geyer, submitted, for the complete theory.)

**Value**

an object of class `mcmc`, subclass `morph.metropolis`. This object is a list containing all of the elements from an object returned by `metrop`, plus at least the following components:

- morph
- morph.final

**References**


**See Also**

`metrop`, `morph`.

**Examples**

```r
out <- morph.metrop(function(x) dt(x, df=3, log=TRUE), 0, blen=100, nbatch=100, morph=morph(b=1))
# change the transformation.
out <- morph.metrop(out, morph=morph(b=2))
out$accept
# accept rate is high, increase the scale.
out <- morph.metrop(out, scale=4)
# close to 0.20 is about right.
out$accept
```

---

**olbm**

*Overlapping Batch Means*

**Description**

Variance of sample mean of time series calculated using overlapping batch means.

**Usage**

```r
olbm(x, batch.length, demean = TRUE)
```
Arguments

- **x**: a matrix or time series object. Each column of x is treated as a scalar time series.
- **batch.length**: length of batches.
- **demean**: when **demean** = **TRUE** (the default) the sample mean is subtracted from each batch mean when estimating the variance. Using **demean** = **FALSE** would essentially assume the true mean is known to be zero, which might be useful in a toy problem where the answer is known.

Value

The estimated variance of the sample mean.

See Also

- **ts**

Examples

```r
h <- function(x) if (all(x >= 0) & sum(x) <= 1) return(1) else return(-Inf)
out <- metrop(h, rep(0, 5), 1000)
out <- metrop(out, scale = 0.1)
out <- metrop(out, nbatch = 1e4)
foo <- olbm(out$batch, 150)
# monte carlo estimates (true means are same by symmetry)
apply(out$batch, 2, mean)
# monte carlo standard errors (true s. d. are same by symmetry)
sqrt(diag(foo))
# check that batch length is reasonable
acf(out$batch, lag.max = 200)
```

Description

Markov chain Monte Carlo (MCMC) for continuous random vectors using parallel or serial tempering, the latter also called umbrella sampling and simulated tempering. The chain simulates $k$ different distributions on the same state space. In parallel tempering, all the distributions are simulated in each iteration. In serial tempering, only one of the distributions is simulated (a random one). In parallel tempering, the state is a $k \times p$ matrix, each row of which is the state for one of the distributions. In serial tempering the state of the Markov chain is a pair $(i, x)$, where $i$ is an integer between 1 and $k$ and $x$ is a vector of length $p$. This pair is represented as a single real vector $c(i, x)$. The variable $i$ says which distribution $x$ is a simulation for.
Usage

temper(obj, initial, neighbors, nbatch, blen = 1, nspac = 1, scale = 1,
        outfun, debug = FALSE, parallel = FALSE, ...)

## S3 method for class 'function'
temper(obj, initial, neighbors, nbatch, blen = 1, nspac = 1, scale = 1,
        outfun, debug = FALSE, parallel = FALSE, ...)

## S3 method for class 'tempering'
temper(obj, initial, neighbors, nbatch, blen = 1, nspac = 1, scale = 1,
        outfun, debug = FALSE, parallel = FALSE, ...)

Arguments

obj either an R function or an object of class "tempering" from a previous run.
If a function, it should evaluate the log unnormalized density \( \log h(i, x) \) of the
desired equilibrium distribution of the Markov chain for serial tempering (the
same function is used for both serial and parallel tempering, see Details below
for further explanation).
If an object of class "tempering", the log unnormalized density function is
obj$ld, and missing arguments of temper are obtained from the corresponding
elements of obj.
The first argument of the log unnormalized density function is the is an R vector
\( c(i, x) \), where \( i \) says which distribution \( x \) is supposed to be a simulation
for. Other arguments are arbitrary and taken from the ... arguments of temper.
The log unnormalized density function should return \(-\infty\) in regions of the state
space having probability zero.

initial for serial tempering, a real vector \( c(i, x) \) as described above. For parallel
 tempering, a real \( k \times p \) matrix as described above. In either case, the initial state
of the Markov chain. Ignored if obj has class "tempering".

neighbors a logical symmetric matrix of dimension \( k \times k \). Elements that are TRUE indicate
jumps or swaps attempted by the Markov chain. Ignored if obj has class "tempering".

nbatch the number of batches.

blen the length of batches.

nspac controls the proposal step size for real elements of the state vector. For serial
tempering, proposing a new value for the \( x \) part of the state \( (i, x) \). For parallel
tempering, proposing a new value for the \( x_i \) part of the state \((x_1, \ldots, x_k)\). In
either case, the proposal is a real vector of length \( p \). If scalar or vector, the
proposal is \( x + \text{scale} \times \text{z} \) where \( x \) is the part \( x \) or \( x_i \) of the state the proposal
may replace. If matrix, the proposal is \( x + \text{scale} \times \times \text{z} \). If list, the length
must be \( k \), and each element must be scalar, vector, or matrix, and operate as
described above. The \( i \)-th component of the list is used to update \( x \) when the
state is \((i, x) \) or \( x_i \) otherwise.
outfun controls the output. If a function, then the batch means of outfun(state, ...) are returned. The argument state is like the argument initial of this function. If missing, the batch means of the real part of the state vector or matrix are returned, and for serial tempering the batch means of a multivariate Bernoulli indicating the current component are returned.

debug if TRUE extra output useful for testing.

parallel if TRUE does parallel tempering, if FALSE does serial tempering. Ignored if obj has class “tempering”.

... additional arguments for obj or outfun.

## Details

Serial tempering simulates a mixture of distributions of a continuous random vector. The number of components of the mixture is $k$, and the dimension of the random vector is $p$. Denote the state $(i, x)$, where $i$ is a positive integer between 1 and $k$, and let $h(i, x)$ denote the unnormalized joint density of their equilibrium distribution. The logarithm of this function is what obj or obj$lud$ calculates. The mixture distribution is the marginal for $x$ derived from the equilibrium distribution $h(i, x)$, that is,

$$ h(x) = \sum_{i=1}^{k} h(i, x) $$

Parallel tempering simulates a product of distributions of a continuous random vector. Denote the state $(x_1, \ldots, x_k)$, then the unnormalized joint density of the equilibrium distribution is

$$ h(x_1, \ldots, x_k) = \prod_{i=1}^{k} h(i, x_i) $$

The update mechanism of the Markov chain combines two kinds of elementary updates: jump/swap updates (jump for serial tempering, swap for parallel tempering) and within-component updates. Each iteration of the Markov chain one of these elementary updates is done. With probability 1/2 a jump/swap update is done, and with probability 1/2 a with-component update is done.

Within-component updates are the same for both serial and parallel tempering. They are “random-walk” Metropolis updates with multivariate normal proposal, the proposal distribution being determined by the argument scale. In serial tempering, the $x$ part of the current state $(i, x)$ is updated preserving $h(i, x)$. In parallel tempering, an index $i$ is chosen at random and the part of the state $x_i$ representing that component is updated, again preserving $h(i, x)$.

Jump updates choose uniformly at random a neighbor of the current component: if $i$ indexes the current component, then it chooses uniformly at random a $j$ such that neighbors$[i, j] == TRUE$. It then does does a Metropolis-Hastings update for changing the current state from $(i, x)$ to $(j, x)$.

Swap updates choose a component uniformly at random and a neighbor of that component uniformly at random: first an index $i$ is chosen uniformly at random between 1 and $k$, then an index $j$ is chosen uniformly at random such that neighbors$[i, j] == TRUE$. It then does does a Metropolis-Hastings update for swapping the states of the two components: interchanging $x_i$ and $x_j$ while preserving $h(x_1, \ldots, x_k)$.

The initial state must satisfy lud(initial, ...) > -Inf for serial tempering or must satisfy lud(initial[i, ], ...) > -Inf for each i for parallel tempering, where lud is either obj or obj$lud$. That is, the initial state must have positive probability.
Value

an object of class "mcmc", subclass "tempering", which is a list containing at least the following components:

- **batch**: the batch means of the continuous part of the state. If `outfun` is missing, an `nbatch` by `k` by `p` array. Otherwise, an `nbatch` by `m` matrix, where `m` is the length of the result of `outfun`.
- **ibatch**: (returned for serial tempering only) an `nbatch` by `k` matrix giving batch means for the multivariate Bernoulli random vector that is all zeros except for a 1 in the `i`-th place when the current state is `(i, x)`.
- **acceptx**: fraction of Metropolis within-component proposals accepted. A vector of length `k` giving the acceptance rate for each component.
- **accepti**: fraction of Metropolis jump/swap proposals accepted. A `k` by `k` matrix giving the acceptance rate for each allowed jump or swap component. NA for elements such that the corresponding elements of `neighbors` is FALSE.
- **initial**: value of argument `initial`.
- **final**: final state of Markov chain.
- **initial.seed**: value of `Random.seed` before the run.
- **final.seed**: value of `Random.seed` after the run.
- **time**: running time of Markov chain from `system.time()`.
- **lud**: the function used to calculate log unnormalized density, either `obj` or `obj$lud` from a previous run.
- **nbatch**: the argument `nbatch` or `obj$nbatch`.
- **blen**: the argument `blen` or `obj$blen`.
- **nspac**: the argument `nspac` or `obj$nspac`.
- **outfun**: the argument `outfun` or `obj$outfun`.

Description of additional output when `debug` = TRUE can be found in the vignette `debug`, which is shown by `vignette("debug", "mcmc")`.

Warning

If `outfun` is missing, then the log unnormalized density function can be defined without a...argument and that works fine. One can define it starting `ludfun <- function(state)` and that works or `ludfun <- function(state, foo, bar)`, where `foo` and `bar` are supplied as additional arguments to `temper` and that works too.

If `outfun` is a function, then both it and the log unnormalized density function can be defined without...arguments if they have exactly the same arguments list and that works fine. Otherwise it doesn’t work. Define these functions by

```r
ludfun <- function(state, foo)
outfun <- function(state, bar)
```

and you get an error about unused arguments. Instead define these functions by
ludfun <- function(state, foo, ...)
outfun <- function(state, bar, ...)

and supply foo and bar as additional arguments to temper, and that works fine.
In short, the log unnormalized density function and outfun need to have ... in their arguments list to be safe. Sometimes it works when ... is left out and sometimes it doesn’t.

Of course, one can avoid this whole issue by always defining the log unnormalized density function and outfun to have only one argument state and use global variables (objects in the R global environment) to specify any other information these functions need to use. That too follows the R way. But some people consider that bad programming practice.

A third option is to define either or both of these functions using a function factory. This is demonstrated in the vignette for this package named demo, which is shown by vignette("demo", "mcmc").

Philosophy of MCMC

This function follows the philosophy of MCMC explained the introductory chapter of the Handbook of Markov Chain Monte Carlo (Geyer, 2011a) and in the chapter of that book on tempering and related subjects (Geyer, 2011b). See also the section on philosophy of metrop.

Tuning

The scale argument must be adjusted so that the acceptance rate for within-component proposals (component acceptx of the result returned by this function) is not too low or too high to get reasonable performance. The log unnormalized density function must be chosen so that the acceptance rate for jump/swap proposals (component accepti of the result returned by this function) is not too low or too high to get reasonable performance. The former is a vector and the latter is a matrix, and all these rates must be adjusted to be reasonable.

The rates in in accepti are influenced by the number of components of the tempering mixture distribution, by what those components are (how far apart they are in some unspecified metric on probability distributions), and by the chosen normalizing constants for those distributions.

For examples of tuning tempering, see Geyer (2011b) and also the vignette of this package shown by vignette("bfst", "mcmc"). The help for R function metrop also gives advice on tuning its sampler, which is relevant for tuning the acceptx rates.

See also Geyer (1991) and Geyer and Thompson (1995) for the general theory of tuning parallel and serial tempering.

References


Examples

```r
d <- 9
witch.which <- c(0.1, 0.3, 0.5, 0.7, 1.0)
ncomp <- length(witch.which)

neighbors <- matrix(FALSE, ncomp, ncomp)
neighbors[ row(neighbors) == col(neighbors) + 1 ] <- TRUE
neighbors[ row(neighbors) == col(neighbors) - 1 ] <- TRUE

ludfun <- function(state, log.pseudo.prior = rep(0, ncomp)) {
  stopifnot(is.numeric(state))
  stopifnot(length(state) == d + 1)
  icomp <- state[1]
  stopifnot(icomp == as.integer(icomp))
  stopifnot(1 <= icomp & icomp <= ncomp)
  stopifnot(is.numeric(log.pseudo.prior))
  stopifnot(length(log.pseudo.prior) == ncomp)
  theta <- state[-1]
  if (any(theta > 1.0)) return(-Inf)
  bnd <- witch.which[icomp]
  lpp <- log.pseudo.prior[icomp]
  if (any(theta > bnd)) return(lpp)
  return(-d * log(bnd) + lpp)
}

# parallel tempering
thetas <- matrix(0.5, ncomp, d)
out <- temper(ludfun, initial = thetas, neighbors = neighbors, nbatch = 20,
             blen = 10, nspac = 5, scale = 0.56789, parallel = TRUE, debug = TRUE)

# serial tempering
theta.initial <- c(1, rep(0.5, d))
# log pseudo prior found by trial and error
qux <- c(0, 9.179, 13.73, 16.71, 20.56)

out <- temper(ludfun, initial = theta.initial, neighbors = neighbors,
              nbatch = 50, blen = 30, nspac = 2, scale = 0.56789,
              parallel = FALSE, debug = FALSE, log.pseudo.prior = qux)
```
Index

*Topic **datasets**
  - foo, 2
  - logit, 4

*Topic **misc**
  - metrop, 5
  - morph, 9
  - morph.metrop, 12
  - temper, 15

*Topic **ts**
  - initseq, 2
  - olbm, 14

foo, 2

initseq, 2

logit, 4

metrop, 3, 4, 5, 12–14, 19
morph, 9, 12–14
morph.metrop, 6, 9, 11, 12

olbm, 14

temper, 8, 9, 15
ts, 15