Package 'overture'

March 15, 2019

Type Package

Title Tools for Writing MCMC

Version 0.2-0

Description Simplifies MCMC setup by automatically looping through sampling functions and saving the results. Reduces the memory footprint of running MCMC and saves samples to disk as the chain runs. Allows samples from the chain to be analyzed while the MCMC is still running. Provides functions for commonly performed operations such as calculating Metropolis acceptance ratios and creating adaptive Metropolis samplers. References: Roberts and Rosenthal (2009) <doi:10.1198/jcgs.2009.06134>.

License LGPL-3

URL https://github.com/kurtis-s/overture

BugReports https://github.com/kurtis-s/overture/issues

Encoding UTF-8

LazyData true

RoxygenNote 6.1.0

Suggests testthat, mockery, covr

Imports bigmemory

NeedsCompilation no

Author Kurtis Shuler [aut, cre]

Maintainer Kurtis Shuler <kurtis.s.1122+CRAN@gmail.com>

Repository CRAN

Date/Publication 2019-03-15 06:40:09 UTC

R topics documented:

`AcceptProp` *Determine if a Metropolis–Hastings step should be accepted*

Description

`AcceptProp` is a utility function to determine if a proposal should be accepted in a Metropolis or Metropolis-Hastings step.

Usage

```
AcceptProp(log.curr, log.prop, log.curr.to.prop = 0,
  log.prop.to.curr = 0)
```

Arguments

`log.curr`	log density of the target at the current value, $log(P(x))$
`log.prop`	log density of the target at the proposed value, $log(P(x'))$
`log.curr.to.prop`	log of transition distribution from current value to proposed value, $log(g(x'\|x))$
`log.prop.to.curr`	log of transition distribution from proposed value to current value, $log(g(x\|x'))$

Details

The function uses the Metropolis choice for a Metropolis/Metropolis-Hastings sampler, which accepts a proposed value x' with probability

$$A(x', x) = min(1, P(x')/P(x)g(x|x')/g(x'|x))$$

where $P(x)$ is the target distribution and $g(x'|x)$ is the proposal distribution.

Value

`TRUE/FALSE` for whether the proposal should be accepted or rejected, respectively

Examples

```
# Sample from triangular distribution P(x) = -2x + 2 ----------------------
# Target distribution
LogP <- function(x) {
    log(-2*x + 2)
}

# Generate proposals using Beta(1/2, 1/2)
shape1 <- 1/2
shape2 <- 1/2

RProp <- function() { # Draw proposal
    rbeta(1, shape1, shape2)
}

DLogProp <- function(x) { # Log density of proposal distribution
    dbeta(x, shape1, shape2, log=TRUE)
}

SampleX <- function(x) { # Draw once from the target distribution
    x.prop <- RProp()
    if(AcceptProp(LogP(x), LogP(x.prop), DLogProp(x.prop), DLogProp(x))) {
        x <- x.prop
    }

    return(x)
}

# Draw from the target distribution
n.samples <- 10000
samples <- vector(length=n.samples)
x <- 0.5
Mcmc <- InitMcmc(n.samples)
samples <- Mcmc({
    x <- SampleX(x)
})

# Plot the results
hist(samples$x, freq=FALSE, ylim=c(0, 2.5), xlim=c(0, 1), xlab="x")
grid <- seq(0, 1, length.out=500)
lines(grid, exp(LogP(grid)), col="blue")
legend("topright", legend="True density", lty=1, col="blue", cex=0.75)
```

`AcceptProposal` *Determine if a Metropolis–Hastings step should be accepted*

Description

`AcceptProposal` is deprecated. Please use `AcceptProp` instead.

Usage

```
AcceptProposal(log.curr, log.prop, log.curr.to.prop = 0,
  log.prop.to.curr = 0)
```

Arguments

`log.curr`	log density of the target at the current value, $log(P(x))$
`log.prop`	log density of the target at the proposed value, $log(P(x'))$
`log.curr.to.prop`	log of transition distribution from current value to proposed value, $log(g(x'\|x))$
`log.prop.to.curr`	log of transition distribution from proposed value to current value, $log(g(x\|x'))$

Value

`TRUE/FALSE` for whether the proposal should be accepted or rejected, respectively

Examples

```
# Sample from triangular distribution P(x) = -2x + 2 ----------------------
# Target distribution
LogP <- function(x) {
    log(-2*x + 2)
}

# Generate proposals using Beta(1/2, 1/2)
shape1 <- 1/2
shape2 <- 1/2

RProp <- function() { # Draw proposal
    rbeta(1, shape1, shape2)
}

DLogProp <- function(x) { # Log density of proposal distribution
    dbeta(x, shape1, shape2, log=TRUE)
}

SampleX <- function(x) { # Draw once from the target distribution
    x.prop <- RProp()
    if(AcceptProp(LogP(x), LogP(x.prop), DLogProp(x.prop), DLogProp(x))) {
        x <- x.prop
    }

    return(x)
}

# Draw from the target distribution
n.samples <- 10000
samples <- vector(length=n.samples)
x <- 0.5
```

```
Mcmc <- InitMcmc(n.samples)
samples <- Mcmc({
    x <- SampleX(x)
})

# Plot the results
hist(samples$x, freq=FALSE, ylim=c(0, 2.5), xlim=c(0, 1), xlab="x")
grid <- seq(0, 1, length.out=500)
lines(grid, exp(LogP(grid)), col="blue")
legend("topright", legend="True density", lty=1, col="blue", cex=0.75)
```

`Amwg` *Turn a non-adaptive Metropolis sampler into an adaptive Metropolis sampler*

Description

Given a non-adpative sampler of the form f(..., s), `Amwg` will return a function g(...) that automatically adapts the Metropolis proposal standard deviation s to try and acheive a target acceptance rate.

Usage

```
Amwg(f, s, batch.size = 50, target = 0.44, DeltaN)
```

Arguments

`f`	non-adaptive Metropolis sampler of the form f(..., s)
`s`	initial value for the Metropolis proposal SD
`batch.size`	number of iterations before proposal SD is adapted
`target`	target acceptance rate
`DeltaN`	function of the form f(n) which returns the adaptation amount based on the number of elapsed iterations, n

Details

`Amwg` uses the Adaptive Metropolis-Within-Gibbs algorithm from Roberts & Rosenthal (2009), which re-scales the proposal standard deviation after a fixed number of MCMC iterations have elapsed. The goal of the algorithm is to acheive a target acceptance rate for the Metropolis step. After the n*th* batch of MCMC iterations the log of the proposal standard deviation, $log(s)$, is increased/decreased by $\delta(n)$. $log(s)$ is increased by $\delta(n)$ if the observed acceptance rate is more than the target acceptance rate, or decreased by $\delta(n)$ if the observed acceptance rate is less than the target acceptance rate. `Amwg` keeps track of the the acceptance rate by comparing the previously sampled value from `f` to the next value. If the two values are equal, the proposal is considered to be rejected, whereas if the two values are different the proposal is considered accepted.

DeltaN is set to $\delta(n) = min(0.01, n^{-1/2})$ unless re-specified in the function call. Some care should be taken if re-specifying DeltaN, as the ergodicity of the chain may not be preserved if certain conditions aren't met. See Roberts & Rosenthal (2009) in the references for details.

The proposal standard deviation s can be either a vector or a scalar. If the initial value of s is a scalar, f will be treated as a sampler for a scalar, a random vector, or a joint parameter update. Alternatively, if the dimension of s is equal to the dimension of the parameters returned by f, the individual elements s will be treated as individual proposal standard deviations for the elements returned by f. This functionality can be used, for example, if f samples each of its returned elements individually, updating each element using a Metropolis step. See the examples for an illustration of this use case. In such settings, f should be contructed to receive s as a vector argument.

Value

Adaptive Metropolis sampler function of the form g(...).

References

Gareth O. Roberts & Jeffrey S. Rosenthal (2009) Examples of Adaptive MCMC, Journal of Computational and Graphical Statistics, 18:2, 349-367, doi: 10.1198/jcgs.2009.06134

Examples

```
# Sample from N(1, 2^2) -------------------------------------------------
LogP <- function(x) dnorm(x, 1, 2, log=TRUE) # Target distribution

f <- function(x, s) { # Non-adaptive Metropolis sampler
    x.prop <- x + rnorm(1, 0, s)
    if(AcceptProp(LogP(x), LogP(x.prop))) {
        x <- x.prop
    }

    return(x)
}

s.start <- 0.1
g <- Amwg(f, s.start, batch.size=25)

n.save <- 10000
Mcmc <- InitMcmc(n.save)
y <- 0
x <- 0
samples <- Mcmc({
    y <- f(y, s.start) # Non-adaptive
    x <- g(x) # Adaptive
})

plot(1:n.save, samples$x, ylim=c(-10, 10), main="Traceplots", xlab="Iteration",
     ylab="Value", type='l')
lines(1:n.save, samples$y, col="red")
legend("bottomleft", legend=c("Adaptive", "Non-adaptive"),
       col=c("black", "red"), lty=1, cex=0.8)
```

```
# Overdispersed Poisson ------------------------------------------------
## Likelihood:
## y_i|theta_i ~ Pois(theta_i), i=1,...,n
## Prior:
## theta_i ~ Log-Normal(mu, sigma^2)
## mu ~ Normal(m, v^2), m and v^2 fixed
## sigma^2 ~ InverseGamma(a, b), a and b fixed


SampleSigma2 <- function(theta.vec, mu, a, b, n.obs) {
    1/rgamma(1, a + n.obs/2, b + (1/2)*sum((log(theta.vec) - mu)^2))
}

SampleMu <- function(theta.vec, sigma.2, m, v.2, n.obs) {
    mu.var <- (1/v.2 + n.obs/sigma.2)^(-1)
    mu.mean <- (m/v.2 + sum(log(theta.vec))/sigma.2) * mu.var

    return(rnorm(1, mu.mean, sqrt(mu.var)))
}

LogDTheta <- function(theta, mu, sigma.2, y) {
    dlnorm(theta, mu, sqrt(sigma.2), log=TRUE) + dpois(y, theta, log=TRUE)
}

# Non-adaptive Metropolis sampler
SampleTheta <- function(theta.vec, mu, sigma.2, y.vec, n.obs, s) {
    theta.prop <- exp(log(theta.vec) + rnorm(n.obs, 0, s))

    # Jacobians, because proposals are made on the log scale
    j.curr <- log(theta.vec)
    j.prop <- log(theta.prop)

    log.curr <- LogDTheta(theta.vec, mu, sigma.2, y.vec) + j.curr
    log.prop <- LogDTheta(theta.prop, mu, sigma.2, y.vec) + j.prop
    theta.vec <- ifelse(AcceptProp(log.curr, log.prop), theta.prop, theta.vec)

    return(theta.vec)
}

## Data
y.vec <- warpbreaks$breaks
n.obs <- length(y.vec)

## Setup adaptive Metropolis sampler
s <- rep(1, n.obs)
# s is a vector, so the acceptance rate of each component will be tracked
# individually in the adaptive Metropolis sampler
SampleThetaAdapative <- Amwg(SampleTheta, s)

## Set prior
v.2 <- 0.05
```

```
m <- log(30) - v.2/2
a <- 1
b <- 2

## Initialize parameters
theta.vec <- y.vec
mu <- m

## MCMC
Mcmc <- InitMcmc(10000)
samples <- Mcmc({
    sigma.2 <- SampleSigma2(theta.vec, mu, a, b, n.obs)
    mu <- SampleMu(theta.vec, sigma.2, m, v.2, n.obs)
    theta.vec <- SampleThetaAdapative(theta.vec, mu, sigma.2, y.vec, n.obs)
})
```

`InitMcmc` *Initialize a Markov chain Monte Carlo run*

Description

Eliminates much of the "boilerplate" code needed for MCMC implementations by looping through the samplers and saving the resulting draws automatically.

Usage

```
InitMcmc(n.save, backing.path = NA, thin = 1, exclude = NULL,
  overwrite = FALSE)
```

Arguments

`n.save`	number of samples to take. If `thin=1`, the number of iterations to run the MCMC chain
`backing.path`	`NA` to save the samples in-memory, otherwise directory path where MCMC samples will be saved
`thin`	thinning interval
`exclude`	character vector specifying variables that should not be saved
`overwrite`	TRUE/FALSE indicating whether previous MCMC results should be overwritten

Details

`InitMcmc` returns a function that takes an R expression. The returned function automatically loops through the R expression and saves any numeric assignments, typically MCMC samples, that are made within it. `exclude` specifies assignments that should not be saved. When `exclude` is `NULL`, all the numeric assignments (scalar, vector, matrix, or array) are saved. The dimensions of matrix

and array assignments are not preserved; they are flattened into vectors before saving. Non-numeric assignments are not saved.

The number of iterations for the MCMC chain is determined by `n.save` and `thin`. The desired number of samples to be saved from the target distribution is set by `n.save`, and the chain is thinned according to the interval set by `thin`. The MCMC chain will run for `n.save` x `thin` iterations.

The MCMC samples can be saved either in-memory or on-disk. Unlike saving in-memory, saving on-disk is not constrained by available RAM. Saving on-disk can be used in high-dimensional settings where running multiple MCMC chains in parallel and saving the results in-memory would use up all available RAM. File-backed saving uses `big.matrix`, and the behaviors of that implementation apply when saving on-disk. In particular, `big.matrix` has call-by-reference rather than call-by-value behavior, so care must be taken not to introduce unintended side-effects when modifying these objects. In-memory saving is implemented via `matrix` and has standard R behavior.

When `backing.path` is `NA`, samples will be saved in-memory. To save samples on-disk, `backing.path` should specify the path to the directory where the MCMC samples should be saved. The `big.matrix` `backingfiles` will be saved in that directory, with filenames corresponding to the variable assignment names made in the R expression. Consequently, the assignment names in the R expression must be chosen in such a way that they are compatible as filenames on the operating system. The `big.matrix` `descriptorfiles` are also named according to the variable assignment names made in the R expression, but with a ".desc" suffix.

By default, `InitMcmc` will not overwrite the results from a previous file-backed MCMC. This behavior can be overridden by specifying `overwrite=TRUE` in `InitMcmc`, or as the second argument to the function returned by `InitMcmc`. See the examples for an illustration. `overwrite` is ignored for in-memory MCMC.

Value

A function that returns a list of either `matrix` or `big.matrix` with the MCMC samples. Each row in the matrices corresponds to one sample from the MCMC chain.

See Also

`bigmemory`

Examples

```
# Beta-binomial ------------------------------------------------------------
## Likelihood:
## x|theta ~ Binomial(n, theta)
## Prior:
## theta ~ Unif(0, 1)

theta.truth <- 0.75
n.obs <- 100
x <- rbinom(1, n.obs, prob=theta.truth)

# Sampling function
SampleTheta <- function() {
    rbeta(1, 1 + x, 1 + n.obs - x)
}
```

```
# MCMC
Mcmc <- InitMcmc(1000)
samples <- Mcmc({
    theta <- SampleTheta()
})

# Plot posterior distribution
hist(samples$theta, freq=FALSE, main="Posterior", xlab=expression(theta))
theta.grid <- seq(min(samples$theta), max(samples$theta), length.out=500)
lines(theta.grid, dbeta(theta.grid, 1 + x, 1 + n.obs - x), col="blue")
abline(v=theta.truth, col="red")
legend("topleft", legend=c("Analytic posterior", "Simulation truth"),
       lty=1, col=c("blue", "red"), cex=0.75)

# Estimating mean with unknown variance ------------------------------------
## Likelihood:
## x|mu, sigma^2 ~ N(mu, sigma^2)
## Prior:
## p(mu) \propto 1
## p(sigma^2) \propto 1/sigma^2

# Simulated data
mu.truth <- 10
sigma.2.truth <- 2
n.obs <- 100
x <- rnorm(n.obs, mu.truth, sqrt(sigma.2.truth))
x.bar <- mean(x)

# Sampling functions
SampleMu <- function(sigma.2) {
    rnorm(1, x.bar, sqrt(sigma.2/n.obs))
}

SampleSigma2 <- function(mu) {
    1/rgamma(1, n.obs/2, (1/2)*sum((x-mu)^2))
}

# MCMC
Mcmc <- InitMcmc(1000, thin=10, exclude="sigma.2")
sigma.2 <- 1 # Initialize parameter
samples <- Mcmc({
    mu <- SampleMu(sigma.2)
    sigma.2 <- SampleSigma2(mu)
})

# Plot posterior distribution
hist(samples$mu, xlab=expression(mu), main="Posterior")
abline(v=mu.truth, col="red")
legend("topleft", legend="Simulation truth", lty=1, col="red", cex=0.75)

# sigma.2 is excluded from saved samples
is.null(samples$sigma.2)
```

```
# Linear regression ----------------------------------------------------
## Likelihood:
## y|beta, sigma^2, x ~ N(x %*% beta, sigma^2 * I)
## Prior:
## p(beta, sigma^2|x) \propto 1/sigma^2

# Simulated data
n.obs <- 100
x <- matrix(NA, nrow=n.obs, ncol=3)
x[,1] <- 1
x[,2] <- rnorm(n.obs)
x[,3] <- x[,2] + rnorm(n.obs)
beta.truth <- c(1, 2, 3)
sigma.2.truth <- 5
y <- rnorm(n.obs, x %*% beta.truth, sqrt(sigma.2.truth))

# Calculations for drawing beta
l.mod <- lm(y ~ x - 1)
beta.hat <- l.mod$coefficients
xtx.inv <- summary(l.mod)$cov.unscaled
xtx.inv.chol <- chol(xtx.inv)

# Calculations for drawing sigma.2
a.sigma.2 <- (n.obs - length(beta.hat))/2
b.sigma.2 <- (1/2) * t(y - x %*% beta.hat) %*% (y - x %*% beta.hat)

# Draw from multivariate normal
Rmvn <- function(mu, sigma.chol) {
    d <- length(mu)
    c(mu + t(sigma.chol) %*% rnorm(d))
}

SampleBeta <- function(sigma.2) {
    Rmvn(beta.hat, xtx.inv.chol * sqrt(sigma.2))
}

SampleSigma2 <- function() {
    1/rgamma(1, a.sigma.2, b.sigma.2)
}

# MCMC, samples saved on-disk
backing.path <- tempfile()
dir.create(backing.path)
Mcmc <- InitMcmc(1000, backing.path=backing.path)
samples <- Mcmc({
    sigma.2 <- SampleSigma2()
    beta <- SampleBeta(sigma.2)
})

# Plot residuals using predictions made from the posterior mean of beta
y.hat <- x %*% colMeans(samples$beta[,])
plot(y.hat, y-y.hat, xlab="Predicted", ylab="Residual")
```

```
abline(h=0, col="red")

# Overwrite previous results ------------------------------------------------
### Overwrite specified in InitMcmc
backing.path <- tempfile()
dir.create(backing.path)
Mcmc <- InitMcmc(5, backing.path=backing.path, overwrite=TRUE)
samples <- Mcmc({
    x <- 1
})
samples <- Mcmc({
    x <- 2
})
samples$x[,]

### Overwrite specified in the function returned by InitMcmc
backing.path <- tempfile()
dir.create(backing.path)
Mcmc <- InitMcmc(5, backing.path=backing.path, overwrite=FALSE)
samples <- Mcmc({
    x <- 3
})
samples <- Mcmc({
    x <- 4
}, overwrite=TRUE)
samples$x[,]
```

LoadMcmc *Load samples from a file-backed MCMC run*

Description

LoadMcmc loads the samples from a file-backed MCMC run initiated by InitMcmc. The result is a list of big.matrix with all of the parameters that were saved in the MCMC run. Alternatively, the samples for individual parameters can be loaded by using attach.big.matrix to load the corresponding descriptor file, "ParameterName.desc," in the MCMC's backing.path directory.

Usage

```
LoadMcmc(backing.path)
```

Arguments

backing.path	directory path where MCMC samples were saved

Value

list of big.matrix with the MCMC samples

See Also

ToMemory, Peek, attach.big.matrix

Examples

```
# Run a file-backed MCMC
backing.path <- tempfile()
dir.create(backing.path)
Mcmc <- InitMcmc(1000, backing.path=backing.path)
samples <- Mcmc({
    x <- rnorm(1)
})
rm(samples)

# Load the saved samples
loaded.samples <- LoadMcmc(backing.path)
hist(loaded.samples$x[,], main="Samples", xlab="x")
```

Peek *Load samples from a partial MCMC run*

Description

Peek allows the samples from a file-backed MCMC to be loaded in another R session while the MCMC is still in progress. By using Peek, the chain's convergence can be monitored before the MCMC chain has finished running.

Usage

```
Peek(backing.path)
```

Arguments

backing.path	directory path of an in-progress MCMC

Value

list of big.matrix with samples from the partial MCMC run

See Also

InitMcmc, LoadMcmc, big.matrix

Examples

```
SampleSomething <- function() {
    Sys.sleep(0.1)
    rnorm(1)
}

backing.path <- tempfile()
dir.create(backing.path)
print(backing.path)

SlowMcmc <- InitMcmc(1000, backing.path=backing.path)
SlowMcmc({
    x <- SampleSomething()
})

### In another R process, while the MCMC is still running...
samples.so.far <- Peek(backing.path)
samples.so.far$x[,]
```

ToMemory *Converts matrices in a file-backed MCMC to R matrix objects*

Description

ToMemory is a convenience method to load the samples from a file-backed MCMC run into memory. Given a list of big.matrix objects, it will convert them to standard R matrix objects.

Usage

```
ToMemory(samples)
```

Arguments

samples	list of big.matrix objects, typically coming from InitMcmc

Value

list of R matrix objects

See Also

InitMcmc, big.matrix

Examples

```
# Run a file-backed MCMC
backing.path <- tempfile()
dir.create(backing.path)
Mcmc <- InitMcmc(1000, backing.path=backing.path)
samples <- Mcmc({
    x <- rnorm(1)
    y <- rnorm(2)
})

# Convert to standard in-memory R matrices
samples.in.memory <- ToMemory(samples)

is.matrix(samples.in.memory$x)
is.matrix(samples.in.memory$y)
bigmemory::is.big.matrix(samples.in.memory$x)
bigmemory::is.big.matrix(samples.in.memory$y)
```

Index