

Package ‘hobbs’

October 5, 2026

Type Package

Title High Dimensional Bayesian Omnibus Sampler

Version 0.4.5

Description Enables high dimensional statistical modeling using Bayesian inference and provides a probabilistic programming language for high dimensional problems. See Kleinsasser (2026)
<[doi:10.5281/zenodo.22309216](https://doi.org/10.5281/zenodo.22309216)>.

License GPL-3

Encoding UTF-8

SystemRequirements Cargo (Rust's package manager), rustc, C compiler

URL <https://hobbs-dev.github.io/>

BugReports <https://github.com/hobbs-dev/hobbs/issues>

Suggests testthat (>= 3.0.0), rjags, rstan

Config/testthat/edition 3

Config/roxygen2/version 8.1.0

NeedsCompilation yes

Author Mike Kleinsasser [aut, cre]

Maintainer Mike Kleinsasser <mjkleinsa@gmail.com>

Repository CRAN

Date/Publication 2026-10-05 19:50:31 UTC

Contents

hobbs	2
hobbs_build_sampler	6
hobbs_check_sampler	7
hobbs_example_model	8
print.hobbs_run	8
read_hobbs	9
read_hobbs_mean	10

Index	13
--------------	-----------

hobbs

*Compile and sample a hobbs Bayesian model***Description**

`hobbs()` is the main model-fitting interface for **hobbs** (High dimensiOnal Bayesian omniBus Sampler), a probabilistic programming system designed for high-dimensional Bayesian models with sparse or structured computational dependencies. Models are written in a compact C-like language using parameter declarations, reusable code chunks, probability statements, and parameter-local sampling blocks.

The R front end translates the model and R data into optimized C code and compiles it as a shared library. A reusable Rust runtime then performs the MCMC sweeps, sampler adaptation, output, and diagnostics. In block mode, model-specific posterior calculations and deterministic cache updates remain in generated C so that each scalar update can perform only the work that its dependency structure requires.

Usage

```
hobbs(
  model,
  data = NULL,
  samples = 1000L,
  burnin = 500L,
  out = file.path(workdir, "chain.bin"),
  workdir = tempdir(),
  seed = 123456789,
  log_cache = FALSE,
  log_cache_bits = 8L
)
```

Arguments

<code>model</code>	A character string containing hobbs model source or a path to a .c model file. Model source can contain <code>param</code> and <code>dparam</code> declarations, <code>func</code> chunks, <code>block</code> declarations, probability statements, and attached <code>cache/update</code> declarations. A continuous declaration may include <code>sampler=rwmh</code> (the default) or <code>sampler=slice</code> , independently of the optional <code>save=mean</code> output modifier. For example, <code>param u(n, p) save=mean sampler=slice</code> ; uses slice sampling while retaining only the posterior mean. One-based ascending loops written as <code>for (i in 1:N)</code> or <code>for (i = 1:N)</code> are translated to C loops before compilation.
<code>data</code>	Optional named list containing model data. Numeric, integer, and logical scalars, vectors, and matrices are supported. Scalars are exposed by name; vectors and matrices receive one-based function-like accessors such as <code>y(i)</code> and <code>X(i, j)</code> . Vector lengths are exposed as <code><name>_len</code> ; matrix dimensions are exposed as <code><name>_len</code> , <code><name>_nrow</code> , and <code><name>_ncol</code> .
<code>samples</code>	Number of retained post-burn-in samples to save. Default 1000.

burnin	Number of discarded burn-in iterations. RWMH proposal scales and slice widths are adapted throughout burn-in, then frozen. Default 500.
out	Output path for the retained binary chain. Defaults to <code>chain.bin</code> in <code>workdir</code> .
workdir	Working directory used for generated model source, data bindings, compiled libraries, and other temporary build products. Defaults to <code>tempdir()</code> .
seed	RNG seed. A numeric seed may be any exactly representable whole number from 0 through $2^{53} - 1$; a decimal character string may be used for the full unsigned 64-bit seed range.
log_cache	Logical. If TRUE, enable the optional lookup-table approximation for selected repeated log/distribution calculations. Default FALSE because this can slightly perturb the log target.
log_cache_bits	Integer table-resolution setting used when <code>log_cache = TRUE</code> . The default 8 allocates 2^8 mantissa lookup values (about 2 KB for the log table). Larger values increase resolution and cache footprint.

Details

The preferred interface declares model parameters directly in the model source with `param` for continuous parameters and `dparam` for bounded discrete parameters. Parameter and data accessors use one-based indexing, matching R. For example, `param beta(p)`; creates `beta(1)`, ..., `beta(p)`, and a matrix supplied as `X` in data can be read as `X(i, j)`. Continuous declarations use `sampler=rwmh` by default and can instead request scalar slice sampling with `sampler=slice`, e.g. `param u(n, p) sampler=slice`;. The sampler modifier is declaration-local, so RWMH and slice parameters can be mixed in the same model.

Probability statements have the form `value ~ distribution(arguments)`; and add the corresponding log density or log probability to the current block target. Repeated calculations can be placed in `func name() { ... }` declarations. These are reusable code chunks that are expanded at their call sites before C compilation, so they can use indices and local declarations from the surrounding block. Ordinary C expressions, scalar declarations, loops, conditionals, transformations, and direct additions to target can also be used. Temporary `vec` and `mat` declarations are available for vector and matrix calculations used by multivariate distributions.

Value

Invisibly returns an object of class `hobbs_run`. It can be passed to `read_hobbs()` to read retained draws and to `read_hobbs_mean()` when the model contains parameters declared with `save=mean`.

Toolchain

`hobbs` models are compiled at run time, so ordinary use requires a working C compiler and the sampler built during package installation. Rust and Cargo are required only for source installation or an explicit sampler rebuild. Use `hobbs_check_sampler()` to verify the installed executable, and `hobbs_build_sampler()` for an explicit build or rebuild in the user cache. `hobbs()` uses the installed sampler and never builds it automatically.

Parameter-local blocks

A declaration such as

```
block beta(j) {
  beta(j) ~ dnorm(0, 10);
  llk();
}
```

creates one scalar update for each coordinate of `beta`. The block need not evaluate the complete log posterior. It should evaluate exactly the prior, likelihood, and other posterior terms whose values can change when that coordinate changes. Terms that do not depend on the active coordinate may be omitted because they are constant in that coordinate's conditional target.

This is an exact computation, not an approximation, provided that every posterior contribution affected by the update is included. A block is therefore both a sampling declaration and an explicit dependency contract. This lets grouped, sparse, latent-variable, and variable-selection models restrict work to the observations or terms actually affected by an update.

Continuous coordinates use sequential scalar Gaussian random-walk Metropolis updates by default. Adding `sampler=slice` to a continuous parameter declaration selects a stepping-out/shrinkage slice update for every scalar coordinate in that declaration. Bounded discrete coordinates declared with, for example, `dparam z(n, 0, 1)`; are updated by evaluating their local block target over every value in the declared support and drawing from the resulting finite-state conditional distribution.

Persistent deterministic caches

A block can maintain exact deterministic state with attached cache and update declarations. For example, a cached linear predictor can be initialized once and updated after a proposal to `beta(j)` using

```
mu(i) += (proposal(beta(j)) - current(beta(j))) * X(i, j);
```

`proposal(...)` is the proposed scalar value and `current(...)` is the currently accepted value. Cache updates are transactional: the proposed parameter and updated cache are used together to evaluate the block target; on rejection, hobbs restores the previous cache automatically. A cache can be maintained by multiple blocks. Correctness requires that its initializer and every attached update keep the cached quantity algebraically consistent with the current parameter state.

Persistent deterministic caches are exact and are distinct from the optional lookup-table approximation controlled by `log_cache`.

Sampling and adaptation

Each continuous coordinate is updated once per sweep using the sampler named in its parameter declaration. `sampler=rwmh` uses the existing adaptive Gaussian random-walk Metropolis kernel with a coordinate-specific Robbins-Monro scale and target acceptance probability 0.44. `sampler=slice` uses univariate slice sampling with randomized stepping out and interval shrinkage. Slice widths start at 1 and adapt from observed scalar jump distances during burn-in only; the widths are frozen for retained sampling. Bounded discrete parameters continue to use exact finite-state Gibbs updates. All three update types can coexist in one model.

Output and high-dimensional storage

By default, retained draws are written to a binary chain and can be read with `read_hobbs()`. A parameter declaration can include `save=mean`, for example `param u(m, 2) save=mean;`. The parameter remains part of the Markov state and is sampled normally, but only its post-burn-in posterior mean is retained. This separates the dimension of the sampled state from the dimension of the stored chain and can greatly reduce storage for large nuisance parameter arrays. Mean-only parameters do not retain information needed for posterior quantiles or convergence diagnostics; use `read_hobbs_mean()` to read their saved means when mixed with full-chain parameters.

The returned `hobbs_run` object records the generated source and shared library, output paths, model dimensions and parameter names, block metadata, adaptation diagnostics, and process status. It can be passed directly to `read_hobbs()`.

Built-in probability statements

Sampling statements currently include scalar continuous distributions `dnorm`, `normal01`, `normal_sd1`, `dunif`, `dexp`, `dgamma`, `dinvgamma`, `dbeta`, `dcauchy`, `dt`, `dchisq`, `dlnorm`, `dlogis`, `dlaplace`, `dweibull`, `dpareto`, `dhalfnorm`, and `dhalfcauchy`; discrete and linked distributions `dbern`, `bernoulli_logit`, `bernoulli_probit`, `bernoulli_cloglog`, `dbinom`, `binomial_logit`, `dpois`, `poisson_log`, `dnbinom`, and `dnbinom_log`; and multivariate/matrix distributions `dbvn`, `dmvn`, `dwish`, `dinvwish`, and `dlkjcorr2`.

Built-in distributions do not limit the models that can be expressed. Model-specific log-density calculations may be placed in a `func` chunk and added directly to target using ordinary C expressions.

Optional distribution lookup cache

Setting `log_cache = TRUE` enables an optional lookup-table approximation for selected repeated elementary calculations. Unlike persistent deterministic caches, this can perturb the numerical log target slightly and is therefore disabled by default. `log_cache_bits` controls table resolution and the associated memory/speed tradeoff. Use this option only when its approximation is acceptable for the application.

See Also

`read_hobbs()`, `read_hobbs_mean()`, `hobbs_build_sampler()`, `hobbs_check_sampler()`

Examples

```
library(hobbs)

set.seed(1)
n <- 200L
p <- 4L
X <- cbind(1, matrix(rnorm(n * (p - 1L)), nrow = n))
beta_true <- c(0.5, 1, -0.75, 0.25)
sigma_true <- 0.75
y <- as.numeric(X %*% beta_true + rnorm(n, sd = sigma_true))

dat <- list(n = n, p = p, X = X, y = y)
```

```

model <- '
param beta(p);
param logsigma(1);

func llk() {
  double sigma = exp(logsigma(1));
  for (i = 1:n) {
    y(i) ~ dnorm(mu(i), sigma);
  }
}

block beta(j) {
  beta(j) ~ dnorm(0, 10);
  llk();
} cache mu(n) {
  for (i = 1:n) {
    for (k = 1:p) {
      mu(i) += beta(k) * X(i, k);
    }
  }
} update mu(n) {
  for (i = 1:n) {
    mu(i) += (proposal(beta(j)) - current(beta(j))) * X(i, j);
  }
}

block logsigma(1) {
  logsigma(1) ~ dnorm(0, 2);
  llk();
}

'

out <- tempfile("regression", fileext = ".bin")

fit <- hobbs(
  model = model,
  data = dat,
  samples = 2000,
  burnin = 1000,
  seed = 123,
  out = out
)

draws <- read_hobbs(fit)
colMeans(draws[paste0("beta[", seq_len(p), "]")])

```

Description

Returns the sampler built during package installation, or explicitly builds the bundled Rust sampler in the user cache when needed. Rebuilding requires Cargo and Rust. Normal hobbs() calls use the installed sampler directly.

Usage

```
hobbs_build_sampler(rebuild = FALSE, quiet = TRUE)
```

Arguments

rebuild	Logical. If TRUE, force a fresh cargo build.
quiet	Logical. If TRUE, suppress build output.

Value

Path to the hobbs executable.

Examples

```
library(hobbs)
hobbs_build_sampler()
```

hobbs_check_sampler	<i>Check the installed hobbs sampler</i>
---------------------	--

Description

Checks whether the sampler installed with the package exists and can be started with --help. Does not require Rust, build a sampler, or modify the user cache. A C compiler is checked separately when compiling a model.

Usage

```
hobbs_check_sampler(quiet = FALSE)
```

Arguments

quiet	Logical. If TRUE, suppress status messages.
-------	---

Value

Invisibly returns TRUE when the installed sampler starts successfully, and FALSE otherwise.

Examples

```
library(hobbs)
hobbs_check_sampler()
```

hobbs_example_model	<i>Write an example C posterior model</i>
---------------------	---

Description

Write an example C posterior model

Usage

```
hobbs_example_model(path = tempfile(fileext = ".c"), batch = TRUE)
```

Arguments

path	Destination path.
batch	Logical. If TRUE, write a batch-capable model.

Value

Path to the written C file.

Examples

```
library(hobbs)
hobbs_example_model()
```

print.hobbs_run	<i>Print a hobbs run</i>
-----------------	--------------------------

Description

Prints a concise summary of a completed hobbs sampling run, including output locations, model dimension, retained samples, warmup settings, and adaptation diagnostics when available.

Usage

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

Arguments

x	An object of class hobbs_run, typically returned by hobbs() .
...	Additional arguments passed to the print method. Currently unused.

Value

Invisibly returns x.

read_hobbs	<i>Read hobbs binary output</i>
------------	---------------------------------

Description

Reads the default binary chain. When an `hobbs_run` uses declaration-level `save=mean`, this returns the full draws for the unmarked parameters. Use [read_hobbs_mean\(\)](#) for the one-row binary of mean-only parameters.

Usage

```
read_hobbs(file, dim = NULL, max_records = NULL, param_names = NULL)
```

Arguments

<code>file</code>	Path to binary chain output.
<code>dim</code>	Number of parameters in the chain. If omitted, it is read from the binary header written by recent hobbs versions.
<code>max_records</code>	Optional maximum number of records to read.
<code>param_names</code>	Optional names for theta columns. Usually supplied automatically when reading a <code>hobbs_run</code> object returned by <code>hobbs()</code> .

Value

A data frame with columns `iter`, `accepted`, `logp`, and the saved parameter columns.

Examples

```
library(hobbs)

set.seed(1)
n <- 200L
p <- 4L
X <- cbind(1, matrix(rnorm(n * (p - 1L)), nrow = n))
beta_true <- c(0.5, 1, -0.75, 0.25)
sigma_true <- 0.75
y <- as.numeric(X %*% beta_true + rnorm(n, sd = sigma_true))

dat <- list(n = n, p = p, X = X, y = y)

model <- '
param beta(p);
param logsigma(1);

func llk() {
  double sigma = exp(logsigma(1));
  for (i = 1:n) {
    y(i) ~ dnorm(mu(i), sigma);
```

```

    }
  }

  block beta(j) {
    beta(j) ~ dnorm(0, 10);
    llk();
  } cache mu(n) {
    for (i = 1:n) {
      for (k = 1:p) {
        mu(i) += beta(k) * X(i, k);
      }
    }
  } update mu(n) {
    for (i = 1:n) {
      mu(i) += (proposal(beta(j)) - current(beta(j))) * X(i, j);
    }
  }

  block logsigma(1) {
    logsigma(1) ~ dnorm(0, 2);
    llk();
  }
}

out <- tempfile("regression", fileext = ".bin")

fit <- hobbs(
  model = model,
  data = dat,
  samples = 2000,
  burnin = 1000,
  seed = 123,
  out = out
)

draws <- read_hobbs(fit)
colMeans(draws[paste0("beta[", seq_len(p), "]")])

```

read_hobbs_mean

Read hobbs posterior-mean output

Description

For declaration-level save=mean, reads the one-record standard hobbs binary written beside the ordinary chain. The saved field is the number of retained draws used in each posterior mean. The older global hobbs(save = "mean") CSV format remains readable for compatibility.

Usage

```
read_hobbs_mean(file, param_names = NULL)
```

Arguments

file An hobbs_run object or path to a mean output file.
 param_names Optional parameter names. Usually supplied automatically.

Value

A one-row data frame with saved, logp, and posterior mean columns.

Examples

```
library(hobbs)

set.seed(1)
n <- 200L
p <- 4L
X <- cbind(1, matrix(rnorm(n * (p - 1L)), nrow = n))
beta_true <- c(0.5, 1, -0.75, 0.25)
sigma_true <- 0.75
y <- as.numeric(X %*% beta_true + rnorm(n, sd = sigma_true))

dat <- list(n = n, p = p, X = X, y = y)

model <- '
param beta(p) save=mean;
param logsigma(1);

func llk() {
  double sigma = exp(logsigma(1));
  for (i = 1:n) {
    y(i) ~ dnorm(mu(i), sigma);
  }
}

block beta(j) {
  beta(j) ~ dnorm(0, 10);
  llk();
} cache mu(n) {
  for (i = 1:n) {
    for (k = 1:p) {
      mu(i) += beta(k) * X(i, k);
    }
  }
} update mu(n) {
  for (i = 1:n) {
    mu(i) += (proposal(beta(j)) - current(beta(j))) * X(i, j);
  }
}

block logsigma(1) {
  logsigma(1) ~ dnorm(0, 2);
  llk();
}
```

```
,  
  
out <- tempfile("regression", fileext = ".bin")  
  
fit <- hobbs(  
  model = model,  
  data = dat,  
  samples = 2000,  
  burnin = 1000,  
  seed = 123,  
  out = out  
)  
  
draws <- read_hobbs(fit)  
draws_mean <- read_hobbs_mean(fit)
```

Index

`hobbs`, [2](#)
`hobbs()`, [8](#)
`hobbs_build_sampler`, [6](#)
`hobbs_build_sampler()`, [3](#), [5](#)
`hobbs_check_sampler`, [7](#)
`hobbs_check_sampler()`, [3](#), [5](#)
`hobbs_example_model`, [8](#)

`print.hobbs_run`, [8](#)

`read_hobbs`, [9](#)
`read_hobbs()`, [3](#), [5](#)
`read_hobbs_mean`, [10](#)
`read_hobbs_mean()`, [3](#), [5](#), [9](#)

`tempdir()`, [3](#)