

Hobbs: A General-Purpose Probabilistic Programming System for High Dimensional Bayesian Data Analysis in R

by Michael Kleinsasser

Abstract General-purpose probabilistic programming makes Bayesian data analysis broadly accessible, but its computational cost can become prohibitive as model dimension grows. The R package *hobbs* (High dimensiOnal Bayesian omniBus Sampler) is a probabilistic programming language and system designed to make high-dimensional Bayesian analysis practical by trading modest syntactic complexity for substantially greater computational efficiency. It allows reusable code for model-specific computational strategies to be expressed directly within the model program. *hobbs* implements adaptive scalar Metropolis-within-Gibbs sampling using parameter-local blocks that evaluate only the posterior terms affected by each proposal. Deterministic caches update linear predictors and other derived quantities incrementally, while optional distribution caches reduce repeated numerical calculations. The R interface translates model programs into compiled C code, while a reusable Rust engine performs sampling and writes posterior draws and diagnostics. Two high-dimensional case studies demonstrate how these features support models with tens of thousands to more than one hundred thousand parameters within a single general modeling framework. Together, these results show that *hobbs* can retain the flexibility of general-purpose probabilistic programming while making large, computationally intensive Bayesian models practical.

1 Introduction

General-purpose probabilistic programming has made Bayesian data analysis accessible across a wide range of applications. Models can be specified at a high level while the system handles posterior computation with relatively little algorithmic input from the user. However, this abstraction can become computationally expensive as model dimension grows. In many probabilistic programming systems, increasing the number of parameters also increases the amount of work repeated for each transition, eventually making otherwise reasonable Bayesian models impractical to fit.

The computational difficulty of a high-dimensional model does not depend only on the number of parameters it contains, but also on how much of the model must be reevaluated when one parameter changes. This distinction is especially important in hierarchical and latent-variable models. A random effect for group g may influence only the observations in that group, yet a generic log-posterior implementation may scan all n observations whenever that effect is proposed. Similarly, changing one regression coefficient alters a linear predictor through a simple rank-one update, while a direct implementation may reconstruct the predictor from all p columns of the design matrix. High-dimensional Bayesian computation can therefore become unnecessarily expensive when software repeatedly recomputes quantities or posterior terms that are unchanged.

hobbs is a general-purpose probabilistic programming language and system for R (R Core Team, 2026) designed around the premise that this computational structure should be expressible directly within the model program. Rather than relying on a single universal optimization strategy, *hobbs* allows model authors to specify reusable code that exploits the dependency structure of a particular model. The resulting program describes not only the probability model, but also how posterior computation can be organized efficiently.

The continuous transition in *hobbs* is scalar Metropolis-within-Gibbs (Tierney, 1994; Gelfand and Smith, 1990) with stochastic-approximation adaptation of the proposal scale (Andrieu and Thoms, 2008). Each continuous coordinate receives a one-dimensional proposal, and its associated block evaluates only the posterior terms that can change under that proposal. Scalar updates require neither gradients nor symbolic differentiation and allow computational dependencies to be expressed at the level of individual parameters. They also make it possible for a reusable Rust sampler to execute very inexpensive transitions against model-specific code compiled to C. The principal tradeoff is that coordinate-wise proposals can mix slowly when posterior parameters are strongly correlated (Roberts and Sahu, 1997). *hobbs* addresses this primarily by making individual transitions sufficiently inexpensive that long chains, including hundreds of thousands or more retained draws when necessary, can remain computationally practical.

Beyond local posterior evaluation, *hobbs* provides explicit mechanisms for avoiding repeated deterministic and numerical work. Linear predictors, residuals, sufficient summaries, and related quantities can be maintained as persistent caches and updated from the difference between proposed

and current parameter values. Accepted proposals commit the staged cache state, whereas rejected proposals automatically restore the previous state. Optional distribution caches can replace selected repeated evaluations of logarithms, inverse links, and probability functions with lookup operations when a controlled numerical approximation is appropriate. Together, these mechanisms reduce proposal cost by exploiting both local dependency structure and repeated computational structure.

This design occupies a different point in the probabilistic programming landscape from several established systems. Stan and PyMC provide broad modeling interfaces centered on automatic differentiation and Hamiltonian Monte Carlo (Carpenter et al., 2017; Abril-Pla et al., 2023). High-performance execution in these systems is supported by optimized numerical backends, vectorized kernels, and, for suitable workloads and backends, accelerator hardware such as GPUs. In contrast, hobbs is designed to obtain competitive performance on conventional CPUs through language and sampler design rather than accelerator hardware, combining compiled C and Rust execution, adaptive scalar Metropolis-within-Gibbs sampling, and explicit exploitation of model dependency structure. JAGS represents models through dependency graphs and applies Gibbs or related component-wise samplers (Plummer, 2003). NIMBLE is a closer conceptual relative in that it compiles statistical models and allows users to configure MCMC algorithms, define custom samplers, and construct model-aware computational procedures (de Valpine et al., 2017). The distinction is that hobbs places proposal-specific computational structure directly in the model specification: programmers can declare which quantities are cached, how those caches are updated under a proposal, and which subsets of the likelihood or prior must be reevaluated. This makes dependency-local computation a first-class part of the modeling language rather than primarily a property of the sampler configuration. For generalized linear mixed models, lme4, glmmTMB, and INLA provide highly optimized estimation or approximate Bayesian inference for supported model classes (Bates et al., 2015; Brooks et al., 2017; Rue et al., 2009). hobbs is not intended to replace these systems. Its purpose is to provide a general language in which model programmers can express and combine model-specific computational strategies while retaining the flexibility to construct models outside predefined families.

The principal contribution of hobbs is therefore its programming and execution model and Rust sampler design, together with the modular way in which these components combine several established computational ideas. Adaptive scalar Metropolis-within-Gibbs is especially compatible with this architecture because its local proposals align naturally with explicit dependency structure, selective likelihood evaluation, and cache updates. These ideas are implemented through three complementary mechanisms:

1. parameter-local blocks that evaluate the posterior terms affected by a scalar proposal;
2. exact deterministic caches that incrementally maintain linear predictors and other derived state; and
3. optional distribution caches that reduce repeated numerical calculations.

These mechanisms provide a framework for expressing model-specific computational strategies directly within a probabilistic program. The objective is not to hide all computational decisions from the user, but to make those decisions concise, reusable, and composable while preserving flexibility in model specification.

The remainder of the article describes the hobbs language, execution model, and correctness requirements before developing these three mechanisms in detail. Their interaction is illustrated first through a random-intercept and random-slope generalized linear mixed model (GLMM) with thousands of parameters. A Gaussian version isolates the computational effects of local posterior evaluation and deterministic caching without introducing approximation. A Bernoulli-logit version then evaluates the speed-accuracy tradeoff introduced by optional distribution caching. A second case study considers high-dimensional sparse variable selection, demonstrating how the same block-and-cache framework can exploit state-dependent as well as observation-level sparsity.

2 The hobbs programming model

This section introduces the core constructs used to specify hobbs models, including parameter declarations, reusable code chunks, parameter-local sampling blocks, and persistent and optional caches.

2.1 Parameter declarations

Parameters are declared at the top level using `param` for continuous parameters and `dparam` for bounded discrete parameters. A continuous declaration has the form

```
,
```

```
param name(nrow);
param name(nrow, ncol);
,
```

where each dimension can be either a positive integer literal or the name of a scalar supplied in the R data list. For example,

```
,
param beta(p);
param u(m, 2);
param logsigma(1);
,
```

declares a vector of p regression coefficients, an m by 2 matrix of random effects, and a scalar log standard deviation. `hobbs` uses one-based indexing in model code, so the elements are referenced as `beta(j)`, `u(j,l)`, and `lowsigma(1)`. This convention matches R and avoids requiring model authors to translate indices when moving between the R data preparation code and the compiled model.

Bounded integer-valued parameters use

```
,
dparam z(n, lower, upper);
,
```

where `lower` and `upper` are inclusive integer bounds. Discrete coordinates are updated by finite enumeration within the corresponding scalar block. Continuous and discrete parameters are stored in a common flattened parameter state internally, but their declared dimensions and names are retained for block construction, output, and diagnostics.

A declaration can include the suffix `save=mean`:

```
,
param u(m, 2) save=mean;
,
```

The parameter is still updated normally during sampling, but `hobbs` retains only its post-warmup posterior mean rather than writing every draw. This is useful for high-dimensional latent variables, such as thousands of group effects, when the full chain is unnecessary or would dominate storage. Parameters without this suffix retain their complete saved chains. Parameter declarations end in semicolons, following C syntax.

2.2 Reusable code chunks

Repeated posterior calculations can be written in a `func` declaration:

```
,
func llk() {
  double sigma = exp(lowsigma(1));

  for (i = 1:n) {
    y(i) ~ dnorm(mu(i), sigma);
  }
}
,
```

The syntax deliberately resembles a zero-argument C function, but a `func` is not compiled as an independently callable function. It is a reusable code chunk that the translator expands directly at each call site. Consequently, functions have no arguments by convention and are declared only as `func name()`. A statement such as

```
,
llk();
,
```

inside a sampling block is replaced by the body of `llk()` before C compilation.

This design choice has two purposes. It avoids function-call overhead in frequently evaluated posterior code, and it allows the expanded code to use the indexing variables and declarations available in the surrounding block. A code chunk can contain ordinary C scalar declarations, transformations, conditionals, loops, and assignments. `hobbs` also provides `vec` and `mat` declarations for temporary vectors and matrices used by multivariate distributions. Because chunks are expanded textually, names should be chosen to avoid collisions within the block into which they are inserted.

Probability statements use the notation

```
,
value(j) ~ distribution(arguments);
,
```

and contribute the corresponding log density or log probability to the block target. The same notation is used for priors and likelihood contributions. Apart from these probability statements and one-based data and parameter accessors, the body remains close to ordinary C. Standard C expressions, control flow, local variables, and user-defined computations remain available without restriction.

2.3 Parameter-local sampling blocks

A central feature of `hobbs` is that each scalar proposal can be paired with only the posterior terms that actually depend on that coordinate. This allows computational work to follow the dependency structure of the model rather than requiring evaluation of the full posterior for every proposal.

Let the complete parameter vector be $\theta = (\theta_j, \theta_{-j})$. For a proposal to coordinate j , suppose the posterior factorizes as

$$\pi(\theta) \propto A_j(\theta_j, \theta_{-j}) B_j(\theta_{-j}). \quad (1)$$

Because B_j is unchanged when only θ_j changes, the Metropolis-Hastings ratio depends only on

$$\frac{A_j(\theta'_j, \theta_{-j})}{A_j(\theta_j, \theta_{-j})}. \quad (2)$$

A block in `hobbs` therefore evaluates $\log A_j$, not necessarily the full log posterior. This remains exact provided that the block includes every prior and likelihood term whose value changes with the proposed coordinate. Terms independent of that coordinate should be omitted because they cancel from the acceptance ratio.

A parameter declaration and sampling block have the following form:

```
model = '
param beta(p);

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

The index j is expanded over the coordinates of `beta`, creating one scalar proposal for each element. The sampling statement contributes the prior density for the proposed coefficient, while `llk()` contributes the likelihood terms affected by that proposal. Scalar parameters use a literal index, as in

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

Array-valued parameters may use multiple indices, as in block `u(j,1)`. A probability statement may also address a range, such as `u(j,1:2) ~ dmvn(...)`, when changing one scalar coordinate

requires reevaluating a joint density involving other elements of the same parameter vector. Discrete declarations instead use finite enumeration and a Gibbs update.

The computational advantage becomes especially clear in grouped models. Consider observations $i = 1, \dots, n$, groups $g = 1, \dots, m$, and group-specific effects u_g . A proposal to u_g changes its prior and the likelihood terms for observations assigned to group g , but not observations in the other groups. A hobbs block can therefore iterate only over the rows belonging to the affected group:

```
,
block u(j, 1) {
  build_sig();
  u(j, 1:2) ~ dmvm(zero2, Sigma_u);
  for (i = gstart(j):gend(j)) {
    y(i) ~ dnorm(eta(i), sigma);
  }
}
```

Although u is indexed by both group and component, the proposal still changes one scalar coordinate at a time. When $u(j, 1)$ is proposed, the bivariate prior for group j is reevaluated because the two components are coupled, while only observations in group j contribute likelihood terms. The number of observation-level densities touched by the proposal therefore falls from n to n_j . In large hierarchical models, reducing the scope of evaluation in this way can be more consequential than reducing the arithmetic cost of an individual density calculation.

Different parameters can naturally have different dependency scopes. Covariance parameters for the random effects may require reevaluating all m random-effect vectors while leaving the observation likelihood untouched. A residual standard deviation in a Gaussian model may affect all n observation densities without affecting the random-effect prior. Fixed effects may influence every observation. These distinct scopes can be represented directly with separate blocks.

In this sense, each block is both a statistical declaration and an explicit dependency contract: the model author specifies the posterior terms affected by a proposal rather than requiring hobbs to reconstruct a complete graphical model at runtime. This is particularly useful when high dimensionality arises from sparse or structured dependencies, because the computational cost of each proposal can follow its actual dependency set rather than the total size of the model.

2.4 Persistent caches and incremental updates

Local likelihood evaluation is not sufficient when each block must rebuild a shared deterministic quantity. Consider the predictor

$$\eta_i = x_i^\top \beta + z_i^\top u_{g_i}, \quad (3)$$

If β_j changes by $\Delta = \beta'_j - \beta_j$, then

$$\eta'_i = \eta_i + \Delta x_{ij}. \quad (4)$$

Rather than recomputing the full predictor after every scalar proposal, hobbs can store it as persistent deterministic state and update it incrementally:

```
,
block beta(j) {
  beta(j) ~ dnorm(0, 10);
  llk();
} cache eta(n) {
  for (i = 1:n) {
    for (k = 1:p) {
      eta(i) += beta(k) * x(i, k);
    }
    for (l = 1:2) {
      eta(i) += u(gid(i), l) * x(i, l);
    }
  }
} update eta(n) {
  for (i = 1:n) {
```

```

    eta(i) += (proposal(beta(j)) - current(beta(j))) * x(i, j);
  }
}

```

The cache declaration initializes `eta` from the current parameter state, while the attached update declaration specifies how the cached value changes when `beta(j)` is proposed. Within an update, `proposal(beta(j))` refers to the proposed scalar value and `current(beta(j))` to the currently accepted value. Reconstructing the predictor directly costs $O(np)$ for each proposed coefficient, whereas the rank-one update costs $O(n)$.

Cache updates are transactional. The proposed parameter and updated cache are used to evaluate the block target. If the proposal is accepted, both become the new state; if it is rejected, `hobbs` restores the previous cache automatically. Additive updates may be reversed algebraically, while more general updates use snapshot-and-restore. The model author therefore specifies only the deterministic forward update.

A cache may be maintained by multiple parameter blocks. For example, a random-effect coordinate updates only the rows belonging to its group:

```

,
block u(j, 1) {
  build_sig();
  u(j, 1:2) ~ dmvn(zero2, Sigma_u);
  pllk();
} update eta(n) {
  for (i = gstart(j):gend(j)) {
    eta(i) += (proposal(u(j, 1)) - current(u(j, 1))) * x(i, 1);
  }
}
,

```

This combines parameter locality with deterministic locality: the block evaluates likelihood terms only for group j , and the cache update touches only the corresponding rows. Shared index symbols connect the proposed parameter coordinate to the affected cache elements.

Persistent caches are intended for exact deterministic state. They do not change the target distribution because the cached quantity is algebraically equivalent to recomputing it from the parameter state. Correctness requires that the initializer and every update remain consistent for all parameters that can change the cached quantity.

2.5 Optional distribution caches

Even after posterior evaluation and deterministic updates are localized, a likelihood may evaluate the same expensive elementary functions millions of times. `hobbs` can optionally replace selected calculations with lookup tables. The current implementation includes a mantissa table for $\log(x)$, tables for Bernoulli logit and probit log probabilities over finite central ranges with exact tail formulas, and an exponential table used by log-link distributions. Integer log-factorials and log-combinations are cached lazily for discrete likelihoods.

Distribution caches differ fundamentally from persistent deterministic caches. A deterministic cache stores an exactly maintained model quantity, whereas a lookup table generally approximates a transcendental function and therefore induces a small perturbation to the log target. For this reason, `log_cache = FALSE` is the default. The table resolution is controlled by `log_cache_bits`; increasing the number of bits improves resolution at the cost of larger lookup tables, which increase memory use and can reduce runtime performance. Exact formulas are used outside supported central ranges rather than extrapolating lookup tables into unstable tails.

The feature is therefore treated as an explicit speed-accuracy tradeoff. The reproduction materials fit the same Bernoulli-logit GLMM using exact calculations and several table resolutions, then compare posterior means, posterior standard deviations, effective sample sizes, and elapsed time. A runtime gain is useful only when the resulting posterior perturbation is negligible relative to Monte Carlo error for the application.

2.6 Complete regression example

The following model combines the language elements in a Gaussian linear regression. The model contains p coefficients and a residual standard deviation represented on the log scale. The likelihood is written once as a reusable code chunk. The coefficient block evaluates the prior for one coefficient and the likelihood for the current proposed predictor. A persistent cache stores the predictor and updates it by a rank-one change. The residual-scale block reuses the same likelihood but does not update the predictor because $\text{logsigma}(1)$ does not affect μ .

```
mod = '
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();
}
```

The declaration `param beta(p)` creates p continuous coordinates, and `block beta(j)` instructs hobbs to update them one at a time. The prior contribution depends only on the proposed coordinate. The likelihood depends on all observations, but it reads the cached predictor rather than rebuilding $x \times \beta$. The cache is initialized once from the complete coefficient vector, after which each proposal changes it by $(\text{proposal}(\beta(j)) - \text{current}(\beta(j))) \times x(i,j)$. The scalar `logsigma(1)` has its own block because changing the residual scale affects the likelihood but not the predictor.

The example illustrates the intended division of responsibility in the language. The model author writes familiar C-like statistical code and identifies the exact computational consequences of each parameter update. More complex models use the same constructs: additional parameter arrays define latent variables, local likelihood chunks restrict evaluation to affected observations, several blocks can maintain a shared cache, and `save=mean` limits storage for large latent states.

3 Sampler and execution architecture

hobbs separates model translation from sampler control. The R front end parses parameter, block, and cache declarations; converts R data into generated C bindings; expands reusable `func` declarations; and generates the routines needed to evaluate parameter-local targets and maintain persistent caches. The resulting C translation unit is compiled with optimization into a shared library that is loaded by a reusable Rust runtime.

For continuous parameters, the generated library also contains block-level sweep kernels that execute sequential scalar Metropolis transitions. A proposed coordinate is written to the parameter

state, any associated deterministic caches are updated, and the corresponding local block target is evaluated. Acceptance commits the proposed parameter and cache state; rejection restores the previous state. Reversible additive cache updates are undone algebraically, whereas more general updates use saved cache state. Performing these operations within generated C keeps the model-specific calculations and the performance-critical scalar transition loop together, while the Rust runtime supplies random variates, controls the block schedule and adaptation, and manages output and diagnostics. Bounded discrete parameters are instead updated by evaluating the local block target over every value in their declared support and drawing from the resulting finite-state conditional distribution.

Because ordinary expressions, control flow, and numerical operations are delegated to the C compiler, the translator only needs to implement constructs specific to the hobbs programming model. The interface between the generated library and the Rust runtime therefore remains independent of the statistical model even though the computational work performed within each block is model specific.

3.1 Warmup and proposal adaptation

Each continuous coordinate is updated with a Gaussian random-walk proposal. At sweep t , coordinate j is proposed as

$$\theta'_j = \theta_j + s_{j,t}Z, \quad Z \sim N(0,1),$$

where $s_{j,t}$ is a coordinate-specific proposal scale. During warmup, hobbs adapts each coordinate-specific proposal scale using a Robbins–Monro stochastic-approximation scheme (Robbins and Monro, 1951; Andrieu and Thoms, 2008). If $A_{j,t}$ is one when the proposal is accepted and zero otherwise,

$$\log s_{j,t+1} = \log s_{j,t} + \gamma_t (A_{j,t} - a^*), \quad \gamma_t = (t + 10)^{-0.6},$$

with default target acceptance probability $a^* = 0.44$, consistent with one-dimensional random-walk optimal-scaling results (Roberts and Rosenthal, 2001). Each continuous coordinate is attempted once per sweep and receives its own adapted scale. By default, adaptation continues through the warmup period and the resulting scales are then held fixed during retained sampling. Bounded discrete parameters use finite-state Gibbs updates and therefore require no proposal-scale adaptation.

4 Using the hobbs package

The Gaussian linear regression described in 2.2.6 is used to walk through a typical analysis using the package and the coda package for some posterior summaries.

```
install.packages("coda")
install.packages("hobbs")
```

Or install the development version from Github.

```
install.packages("pak")
pak::install_github("hobbs-dev/hobbs")
```

4.1 Installation and toolchain

The R package contains the model translator and source for the Rust sampler. A user installs Rust with Cargo and a C compiler, then builds the sampler into the user cache:

```
library(coda)
library(hobbs)

hobbs_check_toolchain()
hobbs_install_sampler()
hobbs_check_sampler()
```

The native toolchain is a deliberate implementation choice, with the cost of a more demanding installation than a pure R package. The package includes diagnostic helpers because compiler and linker failures otherwise appear far from the model code.

4.2 Available distributions

hobbs provides a collection of scalar, discrete, and multivariate probability distributions for model specification. Sampling statements contribute the corresponding log density or log probability to the current block target. Scalar distributions are applied to indexed values such as $x(i)$ or $y(i)$, while multivariate distributions operate on contiguous vector or matrix ranges. Table 1 summarizes the distributions currently available in the model language.

Table 1: Probability distributions currently available in hobbs and their corresponding sampling-statement syntax.

Distribution	Sampling-statement form
Normal	$x(i) \sim \text{dnorm}(\text{mean}, \text{sd});$
Standard normal	$x(i) \sim \text{normal01}();$
Normal with SD 1	$x(i) \sim \text{normal_sd1}(\text{mean});$
Uniform	$x(i) \sim \text{dunif}(\text{min}, \text{max});$
Exponential	$x(i) \sim \text{dexp}(\text{rate});$
Gamma	$x(i) \sim \text{dgamma}(\text{shape}, \text{rate});$
Inverse gamma	$x(i) \sim \text{dinvgamma}(\text{shape}, \text{rate});$
Beta	$x(i) \sim \text{dbeta}(a, b);$
Cauchy	$x(i) \sim \text{dcauchy}(\text{location}, \text{scale});$
Student t	$x(i) \sim \text{dt}(\text{df}, \text{location}, \text{scale});$
Chi-square	$x(i) \sim \text{dchisq}(\text{df});$
Lognormal	$x(i) \sim \text{dlnorm}(\text{meanlog}, \text{sdlog});$
Logistic	$x(i) \sim \text{dlogis}(\text{location}, \text{scale});$
Laplace	$x(i) \sim \text{dlaplace}(\text{location}, \text{scale});$
Weibull	$x(i) \sim \text{dweibull}(\text{shape}, \text{scale});$
Pareto	$x(i) \sim \text{dpareto}(\text{xmin}, \text{alpha});$
Half-normal	$x(i) \sim \text{dhalfnorm}(\text{sd});$
Half-Cauchy	$x(i) \sim \text{dhalfcauchy}(\text{scale});$
Bernoulli	$y(i) \sim \text{dbern}(\text{prob});$
Bernoulli logit	$y(i) \sim \text{bernoulli_logit}(\text{eta});$
Bernoulli probit	$y(i) \sim \text{bernoulli_probit}(\text{eta});$
Bernoulli cloglog	$y(i) \sim \text{bernoulli_cloglog}(\text{eta});$
Binomial	$y(i) \sim \text{dbinom}(\text{size}, \text{prob});$
Binomial logit	$y(i) \sim \text{binomial_logit}(\text{size}, \text{eta});$
Poisson	$y(i) \sim \text{dpois}(\text{lambda});$
Poisson log	$y(i) \sim \text{poisson_log}(\text{eta});$
Negative binomial	$y(i) \sim \text{dnbinom}(\text{size}, \text{prob});$
Negative binomial log-mean	$y(i) \sim \text{dnbinom_log}(\text{eta}, \text{size});$
Bivariate normal, covariance matrix	$v(1:2) \sim \text{dbvn}(\text{mean2}, \text{Sigma2});$
Multivariate normal	$x(1:k) \sim \text{dmvn}(\text{mean}, \text{Sigma});$
Wishart	$W(1:k2) \sim \text{dwish}(\text{scale}, \text{df}, k);$
Inverse Wishart	$W(1:k2) \sim \text{dinvwish}(\text{scale}, \text{df}, k);$
LKJ correlation, 2D	$R(1:4) \sim \text{dlkjcorr2}(\text{eta});$

The distributions in Table 1 are not intended to limit the probability models that can be expressed in hobbs. Custom probability calculations can be defined using a func declaration and reused within sampling blocks. Because function bodies may contain ordinary C expressions, control flow, and numerical calculations, a user can implement a model-specific log density and add its contribution directly to the block target. This allows distributions not included in the built-in library to be incorporated without modifying the sampler itself.

For example, consider a Laplace likelihood in which $\mu(i)$ is the location for observation i and the scale is represented by the parameter $\text{logb}(1)$. A custom log-density contribution can be defined in a reusable func declaration and added directly to the block target:

```
,
func custom_laplace() {
  target += -log(2.0 * b) - fabs(x(i) - mu_i) / b;
}

func llk() {
  double b = exp(logb(1));
```

```

    for (i = 1:n) {
      double mu_i = mu(i);
      custom_laplace();
    }
  }

```

Here, `custom_laplace()` evaluates the Laplace log density for observation i and adds it directly to target. The enclosing `llk()` function applies this contribution across observations. This pattern allows user-defined probability calculations to be combined with the built-in distributions without requiring changes to the hobbs sampler.

4.3 Model, run, and output

A model is supplied as a string or a C file, and data are supplied as a named R list. Scalars, vectors, and matrices receive one-based accessors in generated C. A representative call is:

```

fit = hobbs(
  model = model,
  data = dat,
  samples = 10000,
  warmups = 2000,
  seed = 123,
  out = "glmm.bin"
)

draws = read_hobbs("glmm.bin")

```

The returned object records generated source paths, output paths, model dimensions, adaptation settings, and diagnostic files. Binary storage is used for large chains. A declaration such as

```

,
param u(m, 2) save=mean;
,

```

samples all $2m$ random-effect coordinates normally but saves only their post-warmup means. Global parameters can retain full draws. This separates the dimension of the Markov state from the dimension of the stored chain and is useful when thousands of latent effects are nuisance quantities rather than targets of interval estimation. Mean-only storage does not support quantiles or convergence diagnostics for those coordinates, so it should be chosen only when posterior means are sufficient.

5 Case studies

5.1 Computational environment

All computations were performed on a MacBook Air (Mac15,13) equipped with an Apple M3 processor with eight CPU cores—four performance and four efficiency cores—and 16 GB of unified memory, running macOS 26.6. The examples were run using R 4.5.0 on the aarch64-apple-darwin20 platform. Rust code was compiled using `rustc 1.97.1` and `Cargo 1.97.1` with the stable aarch64-apple-darwin toolchain. C code was compiled using Homebrew Clang 20.1.5 targeting arm64-apple-darwin25.6.0. Reported runtimes are wall-clock times from a single run during rendering of the R Markdown document. The sampler benefits from strong single-thread performance, making Apple M-series systems particularly well suited to this workload.

5.2 GLMM case study

Model and simulated data

The case study uses a random-intercept/random-slope model. Observation i belongs to group g_i and has predictor

Let $x_i = (x_{i1}, \dots, x_{ip})^\top$, with $x_{i1} = 1$ denoting the intercept column. The cached linear predictor used by the implementation is

$$\mu_i = \sum_{k=1}^p \beta_k x_{ik} + u_{g,i,1} x_{i1} + u_{g,i,2} x_{i2}. \quad (5)$$

Thus, because $x_{i1} = 1$, $u_{g,i,1}$ is the group-specific random intercept and $u_{g,i,2}$ is the group-specific random slope associated with the first non-intercept covariate x_{i2} .

The fixed effects have independent priors

$$\beta_k \sim \mathcal{N}(0, 10^2), \quad k = 1, \dots, p. \quad (6)$$

For each group $g = 1, \dots, m$, the two random effects satisfy

$$\begin{pmatrix} u_{g,1} \\ u_{g,2} \end{pmatrix} \sim \mathcal{N}_2 \left[\begin{pmatrix} 0 \\ 0 \end{pmatrix}, \Sigma_u \right], \quad (7)$$

where

$$\Sigma_u = \begin{pmatrix} \tau_0^2 & \rho \tau_0 \tau_1 \\ \rho \tau_0 \tau_1 & \tau_1^2 \end{pmatrix}. \quad (8)$$

The covariance parameters are represented in the implementation through unconstrained transformed parameters,

$$\tau_0 = \exp(\text{logtau0}), \quad \tau_1 = \exp(\text{logtau1}), \quad \rho = \tanh(\text{rho_raw}), \quad (9)$$

with independent priors

$$\text{logtau0} \sim \mathcal{N}(0, 2^2), \quad \text{logtau1} \sim \mathcal{N}(0, 2^2), \quad \text{rho_raw} \sim \mathcal{N}(0, 2^2). \quad (10)$$

For the Gaussian structural benchmark, the residual standard deviation is represented as

$$\sigma = \exp(\text{logsigma}), \quad \text{logsigma} \sim \mathcal{N}(0, 2^2), \quad (11)$$

and the observation model is

$$y_i \mid \mu_i, \sigma \sim \mathcal{N}(\mu_i, \sigma^2), \quad i = 1, \dots, n. \quad (12)$$

The distribution-cache sensitivity analysis replaces Equation (12) with

$$y_i \mid \mu_i \sim \text{Bernoulli} \left\{ \text{logit}^{-1}(\mu_i) \right\}, \quad i = 1, \dots, n, \quad (13)$$

while retaining the same linear predictor and random-effects structure.

The main simulated configuration used $n = 100,000$ observations, $m = 10,000$ balanced groups of 10 observations each, and $p = 6$ fixed effects including the intercept. The five non-intercept covariates were generated independently from standard-normal distributions. Fixed effects were generated as $\beta = (0.5, 1.0, -0.75, 0.5, 0, -0.25)$, with residual standard deviation $\sigma = 0.75$. Group-specific random intercepts and slopes were generated from the bivariate-normal distribution in Equation (7), with $\tau_0 = 1.25$, $\tau_1 = 0.60$, and $\rho = -0.35$. With two random effects per group and four scalar scale/correlation parameters, the Gaussian model contained $p + 2m + 4 = 20,010$ continuous coordinates. The simulation used random seed 123. Posterior inference was based on 200,000 retained MCMC draws following a warmup period of 2,000 iterations. The supplied simulation script records the generating values and uses the same simulated data across all benchmark variants.

The sampler completed 202,000 sweeps, comprising 2,000 adaptive warmup sweeps followed by 200,000 retained posterior draws. Across the 20,010 scalar parameters, this corresponded to approximately 4.04 billion scalar updates. The complete run required approximately 481 seconds, or about 8.0 minutes, corresponding to 420 sweeps per second and 8.40 million scalar updates per second. The overall acceptance rate was 0.444, closely matching the target rate of 0.44, with coordinate-specific acceptance rates ranging from 0.356 to 0.527. Proposal adaptation was performed during the first 2,000 sweeps, with a separate random-walk proposal scale adapted for each continuous coordinate; the resulting scales were then held fixed during retained sampling.

Effective sample sizes for the ten global model parameters (six fixed effects and four residual/covariance parameters) ranged from approximately 806 to 34,794. The intercept had the smallest

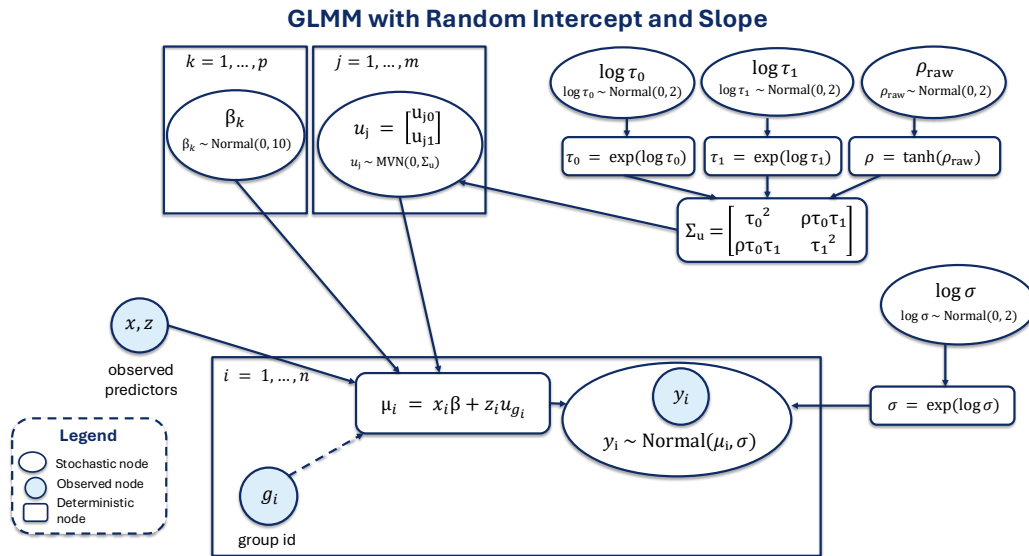


Figure 1: Model-structure diagram for the random-intercept and random-slope Gaussian GLMM. The figure shows the fixed effects, group-specific random effects, covariance parameters, residual scale, deterministic linear predictor, and observation model, together with the indexing structure over predictors, groups, and observations.

effective sample size (806), while the remaining fixed-effect effective sample sizes ranged from approximately 2,910 to 31,726. Effective sample sizes for σ , τ_0 , τ_1 , and ρ were approximately 30,171, 34,794, 21,270, and 23,762, respectively.

The posterior accurately recovered all generating global parameters. Posterior means for the six fixed effects were 0.500, 1.010, -0.750 , 0.497, 0.0005, and -0.251 , respectively, compared with generating values 0.5, 1.0, -0.75 , 0.5, 0, and -0.25 . The generating value of every fixed effect was contained within its 95% equal-tail credible interval. The corresponding 95% intervals were (0.475, 0.524), (0.997, 1.023), (-0.755 , -0.745), (0.492, 0.502), (-0.0046 , 0.0056), and (-0.256 , -0.246).

The residual standard deviation was also accurately recovered, with posterior mean 0.747 and 95% credible interval (0.743, 0.751), compared with the generating value 0.75. The random-intercept standard deviation had posterior mean 1.244 and 95% credible interval (1.226, 1.261), compared with the generating value 1.25, while the random-slope standard deviation had posterior mean 0.607 and 95% credible interval (0.597, 0.617), compared with the generating value 0.60. The random-effect correlation had posterior mean -0.365 and 95% credible interval (-0.384 , -0.345), compared with the generating value -0.35 .

Posterior means of the group-specific random effects also closely tracked their simulated values. Across the 10,000 groups, the correlation between the simulated and posterior-mean random intercepts was 0.980, with root mean squared error (RMSE) 0.246 and mean absolute error (MAE) 0.196. For the random slopes, the corresponding correlation was 0.914, with RMSE 0.246 and MAE 0.193. The empirical standard deviations of the simulated random intercepts and slopes were 1.245 and 0.606, respectively, compared with standard deviations of 1.219 and 0.555 among their posterior means. The somewhat smaller dispersion of the posterior-mean group effects is consistent with shrinkage of the group-specific estimates toward the population mean.

For complete code and results, see the supplemental R Markdown file `glmm-random-intercept-slope.Rmd`.

The complete Gaussian implementation used for this benchmark is

```

,
param beta(p);
param u(m,2) save=mean;
param logsigma(1);
param logtau0(1);
param logtau1(1);
param rho_raw(1);

func llk() {
  double sigma = exp(logsigma(1));

```

```

    for (i = 1:n) {
      y(i) ~ dnorm(mu(i),sigma);
    }
  }

func pll_k() {
  double sigma = exp(logsigma(1));
  for (i = gstart(j):gend(j)) {
    y(i) ~ dnorm(mu(i),sigma);
  }
}

func build_sig() {
  vec zero2(2);
  mat Sigma_u(2,2);

  double tau0 = exp(logtau0(1));
  double tau1 = exp(logtau1(1));
  double rho = tanh(rho_raw(1));

  Sigma_u(1,1) = tau0 * tau0;
  Sigma_u(2,2) = tau1 * tau1;
  Sigma_u(1,2) = rho * tau0 * tau1;
  Sigma_u(2,1) = Sigma_u(1,2);
}

func update_u() {
  for (j = 1:m) {
    u(j,1:2) ~ dmvm(zero2,Sigma_u);
  }
}

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);
    }
    for (l = 1:2) {
      mu(i) += u(gid(i),l) * x(i,l);
    }
  }
} update mu(n) {
  for (i = 1:n) {
    mu(i) += (proposal(beta(j)) - current(beta(j))) * x(i,j);
  }
}

block u(j,l) {
  build_sig();
  u(j,1:2) ~ dmvm(zero2,Sigma_u);
  pll_k();
} update mu(n) {
  for (i = gstart(j):gend(j)) {
    mu(i) += (proposal(u(j,l)) - current(u(j,l))) * x(i,l);
  }
}

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

```

```

block logtau0(1) {
  logtau0(1) ~ dnorm(0,2);
  build_sig();
  update_u();
}

block logtau1(1) {
  logtau1(1) ~ dnorm(0,2);
  build_sig();
  update_u();
}

block rho_raw(1) {
  rho_raw(1) ~ dnorm(0,2);
  build_sig();
  update_u();
}

```

Other GLMM families

The same dependency structure supports Bernoulli, binomial, Poisson, negative-binomial, Gamma, lognormal, Weibull, exponential, beta, Student-*t*, Laplace, logistic, and other observation distributions included in the package header. Only the observation density and any associated scale or shape blocks change. The supplied supplementary R Markdown file contains complete model variants. This is where the scalar/local design is intended to provide generality: the user can change distributions and posterior terms without implementing a new sampler class, while retaining local blocks and cache rules when the dependency pattern is unchanged.

5.3 Computational consequences and benchmark design

Dominant work per sweep

Let n_g be the number of observations in group g , with $\sum_g n_g = n$, and treat the random-effect dimension as fixed at two. Table 2 summarizes dominant predictor and likelihood work. Constants and the small number of global hyperparameter blocks are suppressed.

Table 2: Dominant predictor and log-density work in one scalar sweep of the balanced random-intercept/random-slope model. The expressions describe algorithmic work, not measured elapsed time.

Implementation variant	Fixed-effect work	Group-effect work	Additional work
Full likelihood; rebuild predictor	$O(np^2)$	$O(mnp)$	$O(np + m)$
Local group likelihood; rebuild predictor	$O(np^2)$	$O(np)$	$O(np + m)$
Local group likelihood; linear cache	$O(np)$	$O(n)$	$O(n + m)$

The largest change comes from combining group locality with a linear cache. If every random-effect coordinate triggers a full likelihood and predictor reconstruction, group effects contribute order mnp work per sweep. Restricting each proposal to group g reduces this to order np when the predictor is still rebuilt. Maintaining μ reduces the same part to order n : each observation is touched a constant number of times across all group-effect coordinates. For fixed effects, a linear cache reduces order np^2 predictor reconstruction to order np rank-one updates.

These expressions do not imply that every model will realize the same speedup. Cache writes can become memory-bandwidth limited, small groups reduce the value of further localization, and global

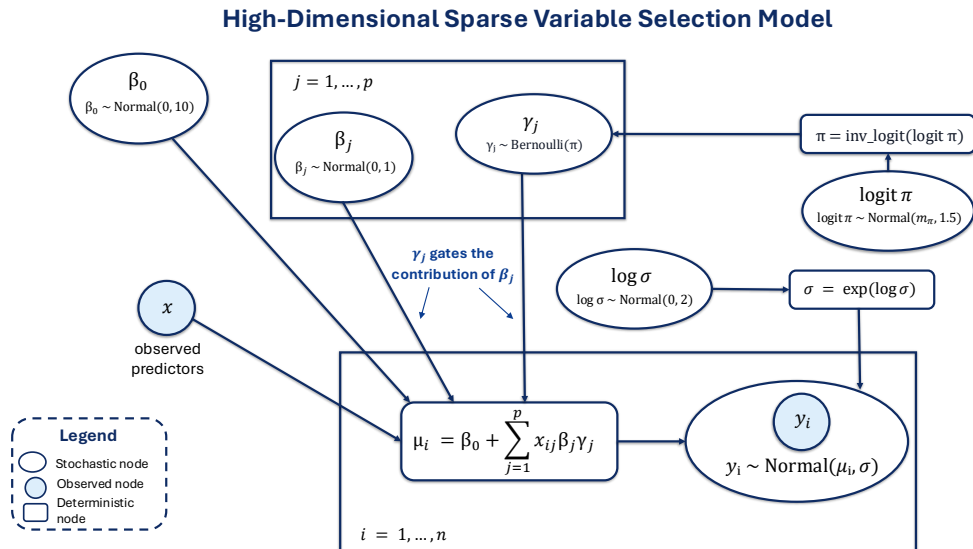


Figure 2: Model-structure diagram for the high-dimensional sparse variable-selection model. The figure shows the intercept and slab coefficients, binary inclusion indicators, inclusion-probability parameter, residual scale, deterministic linear predictor, and Gaussian observation model, with each inclusion indicator gating the contribution of its corresponding predictor.

parameters may dominate when their priors or likelihoods are expensive. The component benchmark therefore measures complete runs rather than reporting only operation counts.

5.4 High-dimensional sparse variable-selection case study

Statistical model and data simulation

The GLMM example derives its computational savings from observation-level locality: changing one group effect alters only the rows assigned to that group. Sparse variable selection presents a different dependency pattern. Every candidate predictor is global because its column can contribute to all observations, but most coefficient proposals do not alter the likelihood when the corresponding predictor is excluded. The example therefore emphasizes conditional dependency, bounded discrete parameters, and exact maintenance of a shared predictor cache.

The intercept and regression coefficients are stored in one parameter vector. This is a compact parameterization rather than a different statistical model: $\text{beta}(1)$ is the always-included intercept, and $\text{beta}(j + 1)$ is the slab coefficient for predictor j . For p candidate predictors,

$$\theta_j = \gamma_j \beta_{j+1}, \quad \gamma_j \mid \pi \sim \text{Bernoulli}(\pi), \quad \beta_{j+1} \sim \mathcal{N}(0, 1), \quad j = 1, \dots, p, \quad (14)$$

and the observation model is

$$\mu_i = \beta_1 + \sum_{j=1}^p \gamma_j \beta_{j+1} x_{ij}, \quad y_i \mid \mu_i, \sigma \sim \mathcal{N}(\mu_i, \sigma^2). \quad (15)$$

The remaining priors are

$$\beta_1 \sim \mathcal{N}(0, 10^2), \quad \log \sigma \sim \mathcal{N}(0, 2^2), \quad \text{logit}(\pi) \sim \mathcal{N}(m_\pi, 1.5^2). \quad (16)$$

For a model with p candidate predictors, the prior location can be defined from a nominal prior model size q_0 as $m_\pi = \text{logit}(q_0/p)$. Because the prior is logit-normal, q_0 is a location-based sparsity target rather than the exact prior expectation of the model size. The simulated example uses $n = 500$, $p = 50,000$, and eight nonzero generating coefficients.

The design matrix was generated from independent standard-normal entries and then centered and scaled columnwise. Eight predictor indices were sampled without replacement, with corresponding nonzero coefficients (2.4, -2.1, 1.8, -1.6, 1.4, -1.2, 1.0, -0.9); all remaining coefficients were set to zero. Responses were generated with intercept ($\beta_1 = 0.5$) and residual standard deviation ($\sigma = 1$), after

which the response vector was centered to have sample mean zero. Posterior inference was based on 10,000 retained MCMC draws following a warmup period of 1,000 iterations. The simulation and MCMC analysis used random seed 123.

The representation in Equation (14) is sometimes described as a product or indicator formulation of spike-and-slab selection. The continuous slab coefficient β_{j+1} remains defined in both inclusion states, while the effective regression coefficient is $\theta_j = \gamma_j \beta_{j+1}$. Consequently, an excluded slab coefficient is sampled from its prior but does not affect the likelihood or predictor cache.

The sampler completed 11,000 sweeps, comprising 1,000 adaptive warmup sweeps followed by 10,000 retained posterior draws. The model contained 100,002 scalar parameters and required approximately 325 seconds, corresponding to 34 sweeps per second and 3.39 million scalar updates per second. The overall acceptance rate for continuous parameters was 0.442, closely matching the target rate of 0.44, with coordinate-specific acceptance rates ranging from 0.332 to 0.558. Trace plots for the residual standard deviation and model size showed stable behavior over the retained draws. Effective sample sizes for the intercept, residual scale, inclusion-probability parameter, and active regression coefficients ranged from approximately 1,371 to 2,431.

The posterior clearly identified all eight generating predictors. Each true active predictor had posterior inclusion probability equal to one, whereas the largest inclusion probability among inactive predictors was 0.205 and nearly all remaining inactive predictors had substantially smaller values. Posterior means of the active effects closely tracked their generating values, with the corresponding credible intervals concentrated around the true effects. The posterior mean model size was 9.95, with median 95% credible interval from 8 to 13, indicating that the sampler consistently retained the eight true predictors while occasionally including a small number of noise variables. The residual standard deviation was also accurately recovered, with posterior mean 1.004 and 95% credible interval $((0.935, 1.073))$. Because the simulated response was centered before fitting, the intercept posterior was centered near zero, with posterior mean 0.055 and 95% credible interval $((-0.036, 0.144))$. The posterior mean inclusion probability was (2.32×10^{-4}) , reflecting the strongly sparse fitted model. For complete code and results, see the supplemental R Markdown file `bayesian-variable-selection.Rmd`.

Complete hobbs model program

The complete model code is shown first and then analyzed block by block. The data contain `p`, the number of candidate predictors, and `p_beta = p + 1`, the length of the combined intercept-and-coefficient vector. `hobbs` parameter extents are named scalar data values, so computing `p_beta` in R keeps the model declaration simple. The scalar `logit_pi_mean` can be computed as `qlogis(prior_model_size / p)`.

```
mod = '
param beta(p_beta);
param logsigma(1);
param logit_pi(1);
dparam gamma(p,0,1);

func y_lpdf() {
  double sigma = exp(logsigma(1));

  for (i = 1:n) {
    y(i) ~ dnorm(mu(i),sigma);
  }
}

block beta(j) {
  if (j == 1) {
    beta(j) ~ dnorm(0,10);
    y_lpdf();
  } else {
    beta(j) ~ dnorm(0,1);

    if (gamma(j-1) != 0) {
      y_lpdf();
    }
  }
}
} cache mu(n) {
  int nactive = 0;
```

```

int active[p];

for (k = 1:p) {
  if (gamma(k) != 0) {
    active[nactive] = k;
    nactive += 1;
  }
}

for (i = 1:n) {
  mu(i) = beta(1);

  if (nactive > 0) {
    for (a = 0:(nactive-1)) {
      int kk = active[a];
      mu(i) += beta(kk+1)*x(i,kk);
    }
  }
}
} update mu(n) {
  double delta = proposal(beta(j)) - current(beta(j));

  if (j == 1) {
    for (i = 1:n) {
      mu(i) += delta;
    }
  } else if (gamma(j-1) != 0) {
    for (i = 1:n) {
      mu(i) += delta * x(i,j-1);
    }
  }
}

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

block gamma(j) {
  double pi = inv_logit(logit_pi(1));

  gamma(j) ~ dbern(pi);
  y_lpdf();
} update mu(n) {
  double delta = proposal(gamma(j)) - current(gamma(j));

  for (i = 1:n) {
    mu(i) += delta*beta(j+1)*x(i,j);
  }
}

block logit_pi(1) {
  double pi = inv_logit(logit_pi(1));
  logit_pi(1) ~ dnorm(logit_pi_mean,1.5);

  for (k = 1:p) {
    gamma(k) ~ dbern(pi);
  }
}

```

The program has three continuous declarations and one bounded discrete declaration. The combined beta vector has length $p + 1$, but only its last p coordinates correspond to selectable predictors. `dparam gamma(p, 0, 1)` therefore has exactly one entry per design-matrix column and

no unused intercept indicator. This offset indexing is deliberate: $\text{gamma}(j)$ controls $\text{beta}(j + 1)$ and column $x(, j)$.

The likelihood is placed in `func y_lpdf()` because it is reused by the intercept, included coefficients, the residual scale, and every discrete indicator. This function always scans the n observations, but the cached value $\text{mu}(i)$ presented to it depends on which block is currently staged. The function therefore describes a common probability contribution without obscuring the different deterministic changes that precede its evaluation.

Block scopes and dominant work

Table 3 summarizes the computational scope of the complete program. Let $q = \sum_{j=1}^p \gamma_j$ denote the current model size.

Table 3: Scalar target and predictor-cache scope in the sparse variable-selection model.

Block	Target	Cache update	Work
<code>beta(1)</code>	Intercept prior and all observations	Add a constant to all $\text{mu}(i)$	$O(n)$
<code>beta(j + 1)</code> , excluded	One slab prior	None	$O(1)$
<code>beta(j + 1)</code> , included	One slab prior and all observations	Add coefficient difference times column j	$O(n)$
<code>gamma(j)</code>	One Bernoulli prior and all observations	Add or remove $\text{beta}(j + 1)$ times column j	$O(n)$
<code>logsigma(1)</code>	Scale prior and all observations	None	$O(n)$
<code>logit_pi(1)</code>	Hyperprior and all indicator priors	None	$O(p)$

Ignoring the two global scalar blocks, one complete sweep requires $O(n + p + nq)$ work for the combined beta vector and $O(np)$ work for the gamma vector. The indicator sweep therefore dominates when all p indicators are updated. Nevertheless, the code expresses two reusable reductions: inactive coefficient transitions are constant-time, and every predictor-changing transition is a rank-one update rather than a complete matrix-vector reconstruction.

The contrast with the GLMM is instructive. In the GLMM, the likelihood itself becomes local because a group effect touches only its group's observations. In variable selection, the likelihood remains global, but whether it is needed depends on the current discrete state. The same block-and-cache language represents both forms of sparsity without requiring a different sampler implementation.

5.5 Position relative to other software

Direct runtime comparisons among Bayesian systems are easy to overinterpret. Stan's NUTS transition, a JAGS component update, a configurable NIMBLE sampler, and an hobbs scalar sweep do different amounts of work and can generate draws with different autocorrelation. Specialized mixed-model packages may optimize or approximate rather than sample the same posterior. Table 4 therefore describes design scope rather than asserting a universal ranking.

Table 4: Qualitative positioning of hobbs relative to general Bayesian and mixed-model software.

System	Primary design	Characteristic strength	Characteristic tradeoff
hobbs	User-declared local scalar targets and exact deterministic caches	Predictable local work and a low-level extensible target language	User must specify locality and cache correctness; native toolchain required

System	Primary design	Characteristic strength	Characteristic tradeoff
Stan / PyMC	Automatic differentiation with Hamiltonian trajectories or NUTS	Strong general-purpose continuous sampling with little manual dependency coding	A gradient evaluation may touch the full model; discrete parameters need other treatment
JAGS	Graph-based Gibbs and related component samplers	Accessible hierarchical model specification and mature Gibbs workflow	Interpreter and scalar-update costs can be high in large models
NIMBLE	Compiled model plus configurable statistical algorithms	Flexible model and sampler programming within R	Greater framework complexity and compilation overhead
lme4 / glmmTMB / INLA	Specialized optimization, Laplace approximation, or latent-Gaussian approximation	Highly optimized inference for supported mixed-model structures	Less general posterior programming or not exact posterior simulation

A fair external benchmark should state the inferential target, parameterization, chain count, warmup policy, compilation treatment, diagnostic threshold, and whether the comparison is based on elapsed time, ESS per second, memory, or all three. The included scripts prioritize an internal ablation because it directly tests the package’s design claims. External comparisons are best added only after equivalent models and diagnostics have been independently verified.

6 Discussion

The hobbs package makes high-dimensional Bayesian computation more efficient by exposing computational dependencies directly in the model program. Parameter-local blocks and persistent caches allow each proposal to update only the posterior terms and deterministic quantities it affects, exploiting both structural and state-dependent sparsity. This trades some automation for computational control and requires correct dependency specification, but enables a reusable scalar Metropolis-within-Gibbs sampler to handle very large models efficiently. Although coordinate-wise sampling can mix slowly for correlated parameters, inexpensive transitions make long chains practical. Overall, the hobbs package provides a middle ground between fully automatic probabilistic programming and hand-written model-specific samplers.

7 Supplementary materials

The submission directory contains the following supplementary materials:

- `supplement/glmm-random-intercept-slope.Rmd`, the first example from the paper.
- `supplement/bayesian-variable-selection.Rmd`, the second example from the paper.

References

O. Abril-Pla, V. Andreani, C. Carroll, L. Dong, C. J. Fonnesbeck, M. Kochurov, R. Kumar, J. Lao, C. C. Luhmann, O. A. Martin, M. Osthege, R. Vieira, T. Wiecki, and R. Zinkov. PyMC: A modern and comprehensive probabilistic programming framework in Python. *PeerJ Computer Science*, 9:e1516, 2023. doi: 10.7717/peerj-cs.1516. [p2]

C. Andrieu and J. Thoms. A tutorial on adaptive MCMC. *Statistics and Computing*, 18(4):343–373, 2008. doi: 10.1007/s11222-008-9110-y. [p1, 8]

D. Bates, M. Maechler, B. Bolker, and S. Walker. Fitting linear mixed-effects models using lme4. *Journal of Statistical Software*, 67(1):1–48, 2015. doi: 10.18637/jss.v067.i01. [p2]

M. E. Brooks, K. Kristensen, K. J. van Benthem, A. Magnusson, C. W. Berg, A. Nielsen, H. J. Skaug, M. Maechler, and B. M. Bolker. glmmTMB balances speed and flexibility among packages for zero-inflated generalized linear mixed modeling. *The R Journal*, 9(2):378–400, 2017. doi: 10.32614/RJ-2017-066. [p2]

- B. Carpenter, A. Gelman, M. D. Hoffman, D. Lee, B. Goodrich, M. Betancourt, M. A. Brubaker, J. Guo, P. Li, and A. Riddell. Stan: A probabilistic programming language. *Journal of Statistical Software*, 76(1):1–32, 2017. doi: 10.18637/jss.v076.i01. [p2]
- P. de Valpine, D. Turek, C. J. Paciorek, C. Anderson-Bergman, D. T. Lang, and R. Bodik. Programming with models: Writing statistical algorithms for general model structures with NIMBLE. *Journal of Computational and Graphical Statistics*, 26(2):403–413, 2017. doi: 10.1080/10618600.2016.1172487. [p2]
- A. E. Gelfand and A. F. M. Smith. Sampling-based approaches to calculating marginal densities. *Journal of the American Statistical Association*, 85(410):398–409, 1990. doi: 10.1080/01621459.1990.10476213. [p1]
- M. Plummer. JAGS: A program for analysis of bayesian graphical models using Gibbs sampling. In *Proceedings of the 3rd International Workshop on Distributed Statistical Computing*, Vienna, Austria, 2003. URL <https://www.r-project.org/conferences/DSC-2003/Proceedings/Plummer.pdf>. [p2]
- R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2026. URL <https://www.R-project.org/>. [p1]
- H. Robbins and S. Monro. A stochastic approximation method. *The Annals of Mathematical Statistics*, 22(3):400–407, 1951. doi: 10.1214/aoms/1177729586. [p8]
- G. O. Roberts and J. S. Rosenthal. Optimal scaling for various Metropolis-Hastings algorithms. *Statistical Science*, 16(4):351–367, 2001. doi: 10.1214/ss/1015346320. [p8]
- G. O. Roberts and S. K. Sahu. Updating schemes, correlation structure, blocking and parameterization for the Gibbs sampler. *Journal of the Royal Statistical Society: Series B*, 59(2):291–317, 1997. doi: 10.1111/1467-9868.00070. [p1]
- H. Rue, S. Martino, and N. Chopin. Approximate bayesian inference for latent gaussian models by using integrated nested laplace approximations. *Journal of the Royal Statistical Society: Series B*, 71(2): 319–392, 2009. doi: 10.1111/j.1467-9868.2008.00700.x. [p2]
- L. Tierney. Markov chains for exploring posterior distributions. *The Annals of Statistics*, 22(4):1701–1728, 1994. doi: 10.1214/aos/1176325750. [p1]

Michael Kleinsasser

University of Michigan

Department of Biostatistics, School of Public Health, Ann Arbor, Michigan, United States

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

mkleinsa@umich.edu