Package: CMCMC (via r-universe)

July 23, 2026

Type Package

Title Contemporaneous Markov Chain Monte Carlo

Version 0.0.1

Description Implements contemporaneous Markov chain Monte Carlo (CMCMC) and interchain adaptive Markov chain Monte Carlo (INCA) samplers of Craiu, Rosenthal and Yang (2009) <doi:10.1198/jasa.2009.tm08393> for targets known up to a normalising constant. The samplers run multiple Metropolis chains in parallel and update proposal covariance estimates using contemporaneous particle groups. Built-in target kernels include multivariate normal, logistic regression, Poisson, Gaussian, Gamma, and hierarchical models, with support for user-provided target kernels. The formula interface glm_cmcmc() fits supported generalized linear models using the built-in kernels. 'CUDA' is used when available, and an 'OpenMP'-enabled CPU backend is available on systems without a 'CUDA' compiler.

License GPL-2 | GPL-3

Encoding UTF-8

Depends R (>= 4.0.0)

SystemRequirements Optional CUDA toolkit and NVIDIA GPU for the CUDA backend; OpenMP for parallel CPU execution.

Suggests knitr, rmarkdown

VignetteBuilder knitr

Config/roxygen2/version 8.0.0

NeedsCompilation yes

Author Ahmad ALQabandi [cre, aut, cph] (ORCID: <https://orcid.org/0009-0006-0055-3846>), Louis Aslett [aut, ths, cph] (ORCID: <https://orcid.org/0000-0003-2211-233X>), Murray Pollock [aut], Gareth Roberts [aut]

Maintainer Ahmad ALQabandi <ahmad.alqabandi@durham.ac.uk>

Config/pak/sysreqs nvidia-cuda-dev

Repository https://cran.r-universe.dev

Date/Publication 2026-07-23 13:10:21 UTC

RemoteUrl https://github.com/cran/CMCMC

RemoteRef HEAD

RemoteSha 09fd0e4cacd3d6dffad8d833ecbb5a1410fc599e

Contents

`cmcmc` *Run a contemporaneous MCMC sampler*

Description

This runs a contemporaneous MCMC sampler and returns a specified number of iterations. The default `backend = "auto"` uses the CUDA backend when a CUDA compiler is available, otherwise it falls back to the built-in CPU backend.

Usage

```
cmcmc(
  init,
  GPUkernel,
  it,
  covit,
  k = if (is.matrix(init)) nrow(init) else length(init) * log(length(init)),
  X = c(),
  Xi = c(),
  saved_iterations = 1,
  initial.sd = 1,
  cflags = "",
  seed = 0,
  device = 0,
  backend = c("auto", "cuda", "cpu"),
  nthreads = NA_integer_,
```

```
  batch = FALSE,
  launch = NULL,
  verbose = FALSE
)
```

Arguments

`init`	the initial population of samples (matrix) or a single starting state, to which normal noise will be added to get an initial population (vector)
`GPUkernel`	target density kernel to evaluate (see 'Target kernels and data layout' in Details). For the CPU backend this can also be an object returned by `load_target_kernel_cpu`.
`it`	total number of MCMC iterations to run
`covit`	number of iterations between covariance updates
`k`	number of particles to use in the population. For `cmcmc()`, this must be even because the population is split into two groups, and it must satisfy `k > 2 * D`, where `D` is the target dimension.
`X`	numeric vector passed to the target kernel as `Xf` (layout depends on `GPUkernel`).
`Xi`	numeric/integer vector passed to the target kernel as `Xi` and coerced to integer type (layout depends on `GPUkernel`).
`saved_iterations`	number of iterations to save: 0 saves all iterations; s>0 saves the last s iterations (i.e. the last s populations).
`initial.sd`	if init is a vector, this is the standard deviation used to seed the initial population based on perturbing init.
`cflags`	optional compiler flags to pass when compiling the GPU kernel.
`seed`	random number generator seed passed through to the sampler.
`device`	GPU device number
`backend`	execution backend. `"auto"` uses CUDA when `nvcc` is available and otherwise uses the CPU backend. `"cuda"` requires a CUDA backend. `"cpu"` uses the built-in C backend.
`nthreads`	number of OpenMP threads for the CPU backend. Ignored when OpenMP support is not available or when using the CUDA backend.
`batch`	logical; if `TRUE`, the CUDA backend runs several Metropolis updates per kernel launch between covariance updates. Batch mode returns only the final population, so `saved_iterations` is set to `1`.
`launch`	optional CUDA launch configuration, usually produced by `cmcmc_autotune`. It must provide positive `threads` and `blocks` values whose product covers at least `k/2` particles. If `NULL`, `cmcmc()` uses a cached tuned configuration when available, otherwise it asks the CUDA occupancy helper for a default launch.
`verbose`	logical; if `TRUE`, emit messages for the selected backend. For CUDA, this includes the launch blocks and threads. For CPU, this includes OpenMP availability, whether more than one OpenMP thread is used, and the CPU thread count.

Details

CMCMC runs a population of K parallel random-walk Metropolis (RWM) chains (in this interface, K is given by the argument `k`). The selected target kernel `GPUkernel` evaluates an unnormalised target density `f(theta)` on a D-dimensional real state space, so the sampler targets `pi(theta)` `proportional to f(theta)`.

Random-walk Metropolis. Given a current state `theta`, RWM proposes `theta_star = theta + xi`, where `xi ~ N_D(0, Sigma_P)`. Since the Gaussian random-walk proposal is symmetric, the acceptance probability is `a(theta, theta_star) = min(1, f(theta_star) / f(theta))`.

Two-group contemporaneous covariance exchange. CMCMC splits the K particles into two equal groups A and B (so K must be even). Let `Theta_t_A` and `Theta_t_B` denote the two `(K / 2) * D` matrices whose rows are the group states at iteration t. At covariance update times, the implementation computes the population empirical covariance within each group, for example:

```
theta_bar_t_B = (2 / K) * sum_{k in B} theta_t_k
C_hat_t_B = (2 / K) * sum_{k in B}
  (theta_t_k - theta_bar_t_B) (theta_t_k - theta_bar_t_B)^T
```

and analogously for group A.

The key CMCMC interaction is that the proposal covariance for one group is taken from the other group. With $s_D = 2.38^2/D$, particles in group A are updated using $\Sigma_{P,A} = s_D \widehat{C}_{t,B}$, and particles in group B are updated using $\Sigma_{P,B} = s_D \widehat{C}_{t,A}$. Between update times, the last covariance factors are reused. The update frequency is controlled by `covit`. The same scaling factor is used by the CPU and CUDA backends.

Cholesky factorisation. To generate Gaussian proposal increments efficiently, the implementation forms a Cholesky factor L such that `Sigma_P = L L^T`, then draws `z ~ N_D(0, I_D)` and sets `xi = z L^T`. The CUDA backend computes this step with NVIDIA cuSOLVER (`cusolverDnSpotrf`), cuRAND, and cuBLAS (`cublasStrmm`). The CPU backend computes the Cholesky factorisation and proposals in C. In the CPU backend, random numbers are generated serially with R's RNG before the OpenMP particle-update loop, so R's global RNG state is not accessed from multiple threads. Since each CMCMC group contains `k/2` particles, the package requires `k/2 > D`. If `k/2 <= D`, the empirical covariance for a D-dimensional target cannot be full rank, so the sampler stops with an error before running.

CUDA batch mode. With `batch = TRUE`, the CUDA backend pre-generates the proposal and uniform random numbers needed until the next covariance update, then runs those Metropolis updates inside one kernel call. This avoids exiting and relaunching the Metropolis kernel at every iteration and can reduce overhead when proposal generation and kernel launches are a meaningful part of the runtime. Batch mode returns only the final population.

Target kernels and data layout. The argument `GPUkernel` selects the target-density kernel. With the CUDA backend, the kernel name must correspond to a built-in `.cu` file in the package `kernels/` directory or to a user CUDA kernel installed with `load_target_kernel`. Compiled CUDA libraries are written to a user cache, not to the installed package directory. Built-in kernels such as `"MVNorm"` and `"Logistic"` are also available in the CPU backend. The auxiliary inputs `X` (numeric) and `Xi` (integer) are passed to the device log-density function `logdens(theta, Xf, Xi)`; their expected layout depends on `GPUkernel`. This package treats `X` as a flat numeric vector (copied to the GPU as single-precision floats) and `Xi` as a flat integer vector (copied as 32-bit integers). For ordinary GLM fitting, prefer `glm_cmcmc`, which constructs the model matrix, response vector, link code, prior

code, and kernel data automatically. Direct use of the built-in GLM kernels through `cmcmc()` is an advanced interface intended for explicit kernel-level control and package testing. With `backend = "cpu"`, the package uses C implementations of the built-in kernels. User-provided CPU kernels can be compiled with `load_target_kernel_cpu` and then passed as `GPUkernel`. User-provided CUDA kernels are not used by the CPU backend.

Currently documented kernels:

- `"MVNorm"` (`inst/kernels/MVNorm.cu`): multivariate normal target with known mean and precision. Here `Xi` is not used and can be empty. `X` must be a numeric vector of length `D + D^2` containing `c(mu, SigmaInv)`, where `mu` is length D and `SigmaInv` is the `D * D` inverse covariance (precision) matrix flattened in column-major order (the default in R). For example:

  ```
  X  <- c(mu, as.numeric(solve(Sigma)))
  Xi <- integer(0)
  ```

- `"Logistic"` (`inst/kernels/Logistic.cu`): logistic regression posterior with an independent N(0, var) prior on coefficients. Let n be the number of observations and D the coefficient dimension. Set `Xi <- as.integer(n)` (so that `Xi[1]` in R, i.e. `Xi[0]` on the device, is n). The numeric vector `X` must be `c(X, y, var)`, where `X` contains the `n * D` design matrix in *row-major* order, `y` is length n with values in {0,1}, and `var` is a single prior variance value (if `var <= 0`, the kernel defaults to 100). In R, a convenient way to form row-major `X` from an `n x D` matrix `Xmat` is `as.numeric(t(Xmat))`:

  ```
  Xi <- as.integer(n)
  X  <- c(as.numeric(t(Xmat)), as.numeric(y), as.numeric(var))
  ```

- `"BinomialGLM"` (`inst/kernels/BinomialGLM.cu`): binomial GLM posterior with an independent N(0, var) prior on coefficients. It uses the same `X` layout as `"Logistic"`, but `Xi` must contain both the number of observations and a link code: `1 = logit` and `2 = probit`. For example:

  ```
  Xi <- as.integer(c(n, 1L))  # logit link
  X  <- c(as.numeric(t(Xmat)), as.numeric(y), as.numeric(var))
  ```

- `"PoissonGLM"` (`inst/kernels/PoissonGLM.cu`): Poisson GLM posterior with an independent N(0, var) prior on coefficients. It uses the same `X` layout as `"Logistic"`, but `y` must contain non-negative counts and `Xi` must contain both the number of observations and a link code: `1 = log`. For example:

  ```
  Xi <- as.integer(c(n, 1L))  # log link
  X  <- c(as.numeric(t(Xmat)), as.numeric(y), as.numeric(var))
  ```

- `"GaussianGLM"` (`inst/kernels/GaussianGLM.cu`): Gaussian GLM posterior with an independent N(0, var) prior on coefficients and a sampled variance parameter `sigma2`. If there are `D` coefficients, the target dimension is `D + 1`, with `theta[D + 1]` storing `sigma2`. The numeric vector `X` must be `c(X, y, var, prior_param1, prior_param2)`, with the design matrix stored in row-major order. `Xi` must contain the number of observations and a link/prior code. For the historical default, `Xi = c(n, 1L)` uses the log link and `sigma2 ~ Uniform(prior_param1, prior_param2)`. More generally, `Xi[2]` can be encoded as `link + 10 * prior_code`, where

link is 1 = log, 2 = inverse, or 3 = identity; prior_code is 1 = Uniform(sigma2), 2 = Uniform(sigma), 3 = Exponential(sigma), or 4 = Half-Cauchy(sigma). Common encoded values are:

```
Xi[2] = 11L  # log link,      Uniform(sigma2)
Xi[2] = 12L  # inverse link,  Uniform(sigma2)
Xi[2] = 13L  # identity link, Uniform(sigma2)
Xi[2] = 23L  # identity link, Uniform(sigma)
Xi[2] = 33L  # identity link, Exponential(sigma)
Xi[2] = 43L  # identity link, Half-Cauchy(sigma)
```

```
Xi <- as.integer(c(n, 13L)) # identity link, Uniform(sigma2)
X  <- c(as.numeric(t(Xmat)), as.numeric(y), as.numeric(var),
        sigma2_lower, sigma2_upper)
```

- "GammaGLM" (inst/kernels/GammaGLM.cu): Gamma GLM posterior with an independent N(0, var) prior on coefficients and a sampled Gamma shape parameter alpha. If there are D coefficients, the target dimension is D + 1, with theta[D + 1] storing alpha. The numeric vector X must be c(X, y, var, prior_param1, prior_param2), with positive responses in y. Xi must contain the number of observations and a link/prior code. For the historical default, Xi = c(n, 1L) uses the log link and alpha ~ Uniform(prior_param1, prior_param2). More generally, Xi[2] can be encoded as link + 10 * prior_code, where link is 1 = log; prior_code is 1 = Uniform(alpha), 2 = Uniform(CV), 3 = Exponential(CV), or 4 = Half-Cauchy(CV), with CV = 1 / sqrt(alpha). The CPU and CUDA backends use the same meanings for these prior codes. For ordinary Gamma GLM fitting, glm_cmcmc uses the CV prior codes; Uniform(alpha) is kept for advanced low-level compatibility. Common encoded values are:

  ```
  Xi[2] = 11L  # log link, Uniform(alpha)
  Xi[2] = 21L  # log link, Uniform(CV)
  Xi[2] = 31L  # log link, Exponential(CV)
  Xi[2] = 41L  # log link, Half-Cauchy(CV)
  ```

  ```
  Xi <- as.integer(c(n, 21L))  # log link, Uniform(CV)
  X  <- c(as.numeric(t(Xmat)), as.numeric(y), as.numeric(var),
          cv_lower, cv_upper)
  ```

- "HierBetaBinomJoint" (inst/kernels/HierBetaBinomJoint.cu): hierarchical Beta–Binomial (joint) posterior. This kernel uses an unconstrained parameterisation with theta = (log(alpha), log(beta), z_1, ..., z_J) where z_j = logit(omega_j). Here X is not used and can be empty. Xi must be the integer vector c(J, n_1..n_J, y_1..y_J). The compile-time dimension must match: D = 2 + J. For example:

  ```
  J  <- length(y)
  X  <- numeric(0)
  Xi <- as.integer(c(J, n, y))
  ```

- "NneWayNormalNC" (inst/kernels/NneWayNormalNC.cu): one-way Normal random-effects model in a non-centred parameterisation. The state is theta = (mu, log(tau), z_1, ..., z_J) with tau > 0 and random effects theta_j = mu + tau z_j. Data are passed via Xi and X as: Xi <- c(J, n_1..n_J) and X <- c(sigma2, ybar_1..ybar_J), where sigma2 is the within-group variance and ybar_j are the group means. Again D = 2 + J must match at compile time. For example:

```
J  <- length(ybar)
Xi <- as.integer(c(J, n))
X  <- c(sigma2, as.numeric(ybar))
```

Value

A numeric matrix with 2 + D columns, where D is the parameter dimension (ncol(init)). The first two columns are metadata:

- column 1: iter (iteration index, 1-based; for saved_iterations=s>0, this will typically run from it-s+1 to it),
- column 2: particle (particle/chain index within the population, 1..K where K=k),
- columns 3..(2+D): the state vector theta.

Let N denote the number of saved iterations. Then the number of rows is K * N. If saved_iterations = 0 (save all), then N = it. If saved_iterations = s > 0, then N = min(s, it).

See Also

cmcmc_autotune, inca, load_target_kernel, load_target_kernel_cpu, cmcmcopt

Examples

```
# Small multivariate normal target. This example uses the CPU backend so it
# can run on machines without CUDA.
D <- 2
K <- 100
set.seed(1)
mu <- c(0.25, -0.5)
Sigma <- matrix(c(1.0, 0.4, 0.4, 1.0), nrow = D)

# MVNorm kernel expects X = c(mu, SigmaInv) with SigmaInv in column-major order.
X <- c(mu, as.numeric(solve(Sigma)))
Xi <- integer(0)
init <- matrix(rnorm(K * D), nrow = K, ncol = D)

samples <- CMCMC::cmcmc(
  init = init,
  GPUkernel = "MVNorm",
  it = 50,
  covit = 5,
  k = K,
  X = X,
  Xi = Xi,
```

```
  saved_iterations = 1,
  seed = 123,
  backend = "cpu"
)

# Posterior sample is the last population (K draws in D dimensions).
dim(samples)
colMeans(samples[, 3:(2 + D), drop = FALSE])
```

`cmcmc_autotune` *Autotune CUDA launch dimensions for a CMCMC target*

Description

`cmcmc_autotune()` tries valid CUDA launch configurations for the given target/kernel configuration, stores the fastest result in a local cache, and returns a sampler function that reuses the tuned configuration.

Usage

```
cmcmc_autotune(
  init,
  GPUkernel,
  covit,
  k = if (is.matrix(init)) nrow(init) else length(init) * log(length(init)),
  X = c(),
  Xi = c(),
  tune_it = NULL,
  repeats = 3L,
  candidates = NULL,
  saved_iterations = 1,
  initial.sd = 1,
  cflags = "",
  seed = 1,
  device = 0,
  batch = FALSE,
  save = TRUE,
  verbose = TRUE
)
```

Arguments

`init`	the initial population of samples (matrix) or a single starting state, to which normal noise will be added to get an initial population (vector)
`GPUkernel`	target density kernel to evaluate (see 'Target kernels and data layout' in Details). For the CPU backend this can also be an object returned by `load_target_kernel_cpu`.

covit	number of iterations between covariance updates
k	number of particles to use in the population. For cmcmc(), this must be even because the population is split into two groups, and it must satisfy k > 2 * D, where D is the target dimension.
X	numeric vector passed to the target kernel as Xf (layout depends on GPUkernel).
Xi	numeric/integer vector passed to the target kernel as Xi and coerced to integer type (layout depends on GPUkernel).
tune_it	number of low-level Metropolis iterations used for each timing run. If NULL, covit is used, matching the covariance block length.
repeats	number of timing repeats per launch candidate.
candidates	optional data frame/list with integer threads and blocks columns/elements. By default cmcmc_autotune() first asks CUDA for an occupancy-based block size, then tries standard CUDA block sizes around that suggestion, with enough blocks to cover at least k / 2 particles.
saved_iterations	number of iterations to save: 0 saves all iterations; s>0 saves the last s iterations (i.e. the last s populations).
initial.sd	if init is a vector, this is the standard deviation used to seed the initial population based on perturbing init.
cflags	optional compiler flags to pass when compiling the GPU kernel.
seed	random number generator seed passed through to the sampler.
device	GPU device number
batch	logical; if TRUE, the CUDA backend runs several Metropolis updates per kernel launch between covariance updates. Batch mode returns only the final population, so saved_iterations is set to 1.
save	logical; if TRUE, save the best launch configuration in the user cache.
verbose	logical; if TRUE, emit tuning progress messages.

Value

A function with arguments it, seed, init, saved_iterations, and .... Calling it runs cmcmc with the tuned launch configuration. The function has launch, occupancy_launch, timings, and tuning_summary attributes.

Examples

```
## Not run:
# Autotuning requires a CUDA-capable machine and performs timing runs.
D <- 4L
K <- 128L
it <- 100L
covit <- 5L

set.seed(1)
mu <- rep(0, D)
```

```
SigmaInv <- diag(D)
X <- c(mu, as.numeric(SigmaInv))
init <- matrix(rnorm(K * D), nrow = K, ncol = D)

tuned <- cmcmc_autotune(
  init = init,
  GPUkernel = "MVNorm",
  covit = covit,
  k = K,
  X = X,
  Xi = integer(0),
  batch = TRUE,
  repeats = 3,
  save = FALSE
)

attr(tuned, "launch")
samples <- tuned(it = it, seed = 1, saved_iterations = 1)
dim(samples)

## End(Not run)
```

`cmcmc_autotune_clear` *Clear cached CMCMC launch tuning results*

Description

Clear cached CMCMC launch tuning results

Usage

```
cmcmc_autotune_clear(all = TRUE)
```

Arguments

`all`	logical; if `TRUE`, remove the full launch-tuning cache.

Value

Invisibly returns `TRUE` if the cache file was removed.

Examples

```
## Not run:
# This removes all locally cached CMCMC launch tuning results.
cmcmc_autotune_clear()

## End(Not run)
```

cmcmc_autotune_list *List cached CMCMC launch tuning results*

Description

Return cached CUDA launch-tuning results.

Usage

```
cmcmc_autotune_list()
```

Value

A data frame of cached launch-tuning results. The cache file path is stored in attribute `"cache_file"`.

Examples

```
cmcmc_autotune_list()
```

cmcmc_clear_kernel_cache *Clear the CMCMC CUDA kernel cache*

Description

Removes runtime CUDA files created by CMCMC. This includes compiled CUDA shared libraries, and optionally user CUDA source kernels installed with `load_target_kernel`. The package does not write these files into the installed package directory; they live under the user cache directory `file.path(tools::R_user_dir("CMCMC", "cache"), "kernels")`, unless `CMCMC_KERNEL_CACHE` is set to a different kernel-cache root.

Usage

```
cmcmc_clear_kernel_cache(all = TRUE)
```

Arguments

`all`	logical. If `TRUE`, remove the full CMCMC CUDA kernel cache, including user CUDA sources installed with `load_target_kernel`. If `FALSE`, remove only compiled CUDA libraries for the current package version.

Details

R does not provide a reliable package uninstall hook that can clean user cache files automatically. To remove runtime CUDA files as part of uninstall cleanup, call `cmcmc_clear_kernel_cache()` before `remove.packages("CMCMC")`.

Value

Invisibly returns the cache path that was removed, or `character(0)` if no cache directory existed.

Examples

```
## Not run:
# Remove all runtime CUDA cache files before uninstalling the package.
cmcmc_clear_kernel_cache()
remove.packages("CMCMC")

## End(Not run)
```

`cmcmcopt` *Set options related to the package*

Description

Set options related to the package

Usage

```
cmcmcopt(PATH = NA, DYLD_LIBRARY_PATH = NA, verbose_compile = NA)
```

Arguments

`PATH`	the path to the CUDA binary directory
`DYLD_LIBRARY_PATH`	the path to the CUDA library directory. On Windows, this directory is also prepended to `PATH` while compiling kernels.
`verbose_compile`	logical; if `TRUE`, prints CUDA kernel compile progress, the `nvcc` command, and compiler output when compiling GPU kernels at runtime. If `FALSE`, successful compilation is quiet.

Details

CMCMC tries to detect CUDA paths automatically when the package is loaded. It checks `CMCMC_CUDA_BIN`, `CMCMC_CUDA_LIB64`, CUDA environment variables such as `CUDA_HOME` and `CUDA_PATH`, and common Linux, macOS, and Windows install locations. Use `cmcmcopt()` only when you need to override the detected paths for the current R session. These overrides are not written to disk. For persistent CUDA paths, set `CMCMC_CUDA_BIN` and `CMCMC_CUDA_LIB64` before loading the package, for example in `~/.Renviron`.

CUDA target kernels are compiled at runtime into a user-writable cache. The default kernel cache root is `file.path(tools::R_user_dir("CMCMC", "cache"), "kernels")`; set `CMCMC_KERNEL_CACHE` before loading or running the package to use a different writable kernel-cache directory. Launch-tuning results are cached separately under `tools::R_user_dir("CMCMC", "cache")`. Use `cmcmc_clear_kernel_cache` to remove runtime CUDA kernel files, for example before uninstalling the package.

Value

Invisibly returns NULL. The function is called for its side effect of updating CMCMC options for the current R session.

Examples

```
# Suppress successful CUDA compile messages for the current R session.
cmcmcopt(verbose_compile = FALSE)
```

---

glm_cmcmc *Fit a GLM with CMCMC*

Description

glm_cmcmc() provides a formula interface for the built-in CMCMC GLM target kernels. It uses model.frame and model.matrix to build the design matrix, then calls cmcmc with the low-level X and Xi layout required by the selected GLM kernel.

Usage

```
glm_cmcmc(
  formula,
  family = stats::binomial(),
  data,
  k,
  it,
  covit,
  prior_var = 100,
  prior.override = list(),
  init = NULL,
  initial.sd = 0.1,
  seed = 0,
  verbose = FALSE
)
```

Arguments

formula	a model formula, as in glm.
family	a family object, family function, or family name. Supported combinations are binomial("logit"), binomial("probit"), poisson("log"), gaussian("identity"), gaussian("log"), gaussian("inverse"), and Gamma("log").
data	an optional data frame containing the variables in formula.

k	number of CMCMC particles. This must satisfy the requirements of cmcmc: it must be even and k / 2 must be larger than the sampled state dimension. For binomial and Poisson models this is the number of coefficients; for Gaussian and Gamma models it is the number of coefficients plus one auxiliary scale/shape parameter.
it	total number of MCMC iterations.
covit	number of iterations between covariance updates.
prior_var	prior variance for each regression coefficient.
prior.override	optional list of family-specific prior settings for Gaussian and Gamma models. See Details for supported fields, defaults, and prior choices.
init	optional initial state vector or particle matrix. If NULL, binomial and Poisson models start from zero coefficients, while Gaussian and Gamma models start from reference glm.fit() coefficients plus the default or supplied auxiliary initial value.
initial.sd	initial standard deviation passed to cmcmc when init is a vector or NULL.
seed	random number generator seed passed to cmcmc.
verbose	logical; if TRUE, print the backend selected by cmcmc. For CUDA this includes launch blocks and threads; for CPU this includes OpenMP availability and thread count.

Details

This version supports the following family and link combinations: binomial("logit"), binomial("probit"), poisson("log"), gaussian("identity"), gaussian("log"), gaussian("inverse"), and Gamma("log"). Character family names are also accepted; "binomial" and "poisson" use their usual default links, "gaussian" uses its usual identity link, and "gamma" uses the supported log link.

Binomial responses must be binary. Numeric and integer binomial responses must contain only 0 and 1. Logical responses are converted to 0/1. Two-level factor responses are supported, with the second factor level treated as the success level. Poisson responses must be non-negative integer counts. Gaussian responses must be finite numeric values. Gamma responses must be positive numeric values.

The sampled coefficient model is

$$y_i \mid \beta \sim F(g^{-1}(x_i^T \beta)),$$

where F is the selected GLM family, with independent coefficient priors

$$\beta_j \sim N(0, prior\_var).$$

For Gaussian models, the coefficient vector is augmented with a sampled variance parameter sigma2. Set prior.override$sigma_prior to one of the following values.

- "uniform_sigma2": $\sigma^2 \sim Uniform$. Uses sigma2_lower, sigma2_upper, and sigma2_init; this is the default bounded baseline on the variance.
- "uniform_sigma": $\sigma \sim Uniform$. Uses sigma_lower, sigma_upper, and sigma2_init; this gives a bounded flat prior on the residual standard deviation.

- "exponential_sigma": tail-calibrated by $P(\sigma > U) = q$. Uses sigma_upper, sigma_tail_prob, and sigma2_init; this is a PC-style shrinkage prior toward smaller residual scale.
- "half_cauchy_sigma": tail-calibrated by $P(\sigma > U) = q$. Uses sigma_upper, sigma_tail_prob, and sigma2_init; this is a heavy-tailed weakly informative sensitivity option.

For the tail-calibrated Gaussian scale priors, sigma_upper is U and sigma_tail_prob is q. The exponential prior uses rate $\lambda = -\log(q)/U$; the half-Cauchy prior uses scale $A = U\tan(\pi q/2)$. Gaussian defaults are sigma_prior = "uniform_sigma2", sigma2_lower = 0, sigma2_upper = 1.05 * var(y), sigma_lower = sqrt(sigma2_lower), sigma_upper = sqrt(sigma2_upper), sigma_tail_prob = 0.05, and sigma2_init equal to the reference glm.fit() residual variance.

For Gamma models, the coefficient vector is augmented with a sampled shape parameter alpha, with $\mathrm{Var}(Y_i) = \mu_i^2/\alpha$. The prior is placed on $CV = 1/\sqrt{\alpha}$, while the sampled column remains alpha. Set prior.override$cv_prior to one of the following values.

- "uniform_cv": $CV \sim Uniform$. Uses cv_lower, cv_upper, and alpha_init; this is the default bounded baseline on relative variation.
- "exponential_cv": tail-calibrated by $P(CV > U) = q$. Uses cv_upper, cv_tail_prob, and alpha_init; this is a PC-style shrinkage prior toward smaller relative variation.
- "half_cauchy_cv": tail-calibrated by $P(CV > U) = q$. Uses cv_upper, cv_tail_prob, and alpha_init; this is a heavy-tailed weakly informative sensitivity option.

For the tail-calibrated Gamma CV priors, cv_upper is U and cv_tail_prob is q. The exponential prior uses rate $\lambda = -\log(q)/U$; the half-Cauchy prior uses scale $A = U\tan(\pi q/2)$. Gamma defaults are cv_prior = "uniform_cv", cv_lower = 0, cv_upper = 1.05 * sqrt(var(y) / mean(y)^2), cv_tail_prob = 0.05, and alpha_init equal to 1 / glm.fit() dispersion. Binomial and Poisson models do not use prior.override, because their conditional variances are determined by β: $p_i(1-p_i)$ for binomial responses and μ_i for Poisson responses.

The wrapper intentionally rejects features that are not supported by the current GLM kernels: grouped binomial responses, observation weights, offsets, and unsupported families. Supported binomial links are "logit" and "probit". The supported Poisson link is "log". Supported Gaussian links are "identity", "log", and "inverse". The supported Gamma link is "log".

glm_cmcmc() calls cmcmc with backend = "auto". When CUDA is available, the built-in GLM kernels use the CUDA backend; otherwise they use the CPU backend. Use cmcmc directly when explicit backend, device, compiler flag, or CUDA launch control is required.

Value

A data frame with class "glm_cmcmc". The columns are iter, particle, and one column for each coefficient in the model matrix. Gaussian models include one additional sampled column, sigma2; Gamma models include one additional sampled column, alpha. The posterior summary and model metadata are stored as attributes and are used by print(), summary(), and coef() methods. coef() returns posterior means for regression coefficients only.

See Also

cmcmc

Examples

```
set.seed(1)
n <- 80
dat <- data.frame(
  y = rbinom(n, 1, 0.5),
  x = rnorm(n)
)

fit <- glm_cmcmc(
  y ~ x,
  data = dat,
  family = binomial(),
  k = 20,
  it = 20,
  covit = 5,
  seed = 1
)

coef(fit)
summary(fit)
```

`inca` *Run an interchain adaptive MCMC sampler*

Description

Runs the interchain adaptive Markov chain Monte Carlo (INCA) strategy of Craiu, Rosenthal and Yang (2009) in its global covariance form. The sampler runs a population of parallel random-walk Metropolis chains and adapts their shared proposal covariance using samples pooled across chains. The default `backend = "auto"` uses the CUDA backend when a CUDA compiler is available, otherwise it falls back to the built-in CPU backend.

Usage

```
inca(
  init,
  GPUkernel,
  it,
  covit,
  burnin = 0,
  k = if (is.matrix(init)) nrow(init) else length(init) * log(length(init)),
  X = c(),
  Xi = c(),
  saved_iterations = 1,
  initial.sd = 1,
  cflags = "",
  seed = 0,
```

```
  device = 0,
  backend = c("auto", "cuda", "cpu"),
  nthreads = NA_integer_,
  verbose = FALSE
)
```

Arguments

`init`	the initial population of samples (matrix) or a single starting state, to which normal noise will be added to get an initial population (vector)
`GPUkernel`	target density kernel to evaluate (see Details). For the CPU backend this can also be an object returned by `load_target_kernel_cpu`.
`it`	total number of MCMC iterations to run
`covit`	number of iterations between covariance updates
`burnin`	number of initial iterations to run using the initial isotropic proposal covariance before applying covariance adaptation. The running empirical covariance is still accumulated from iteration 1 onwards, but it is not used to update the proposal until the burn-in period has completed.
`k`	number of particles to use in the population.
`X`	numeric vector passed to the target kernel as `Xf` (layout depends on `GPUkernel`).
`Xi`	an additional vector which your target log density may depend upon but which is coerced to integer type (layout depends on `GPUkernel`).
`saved_iterations`	number of iterations to save: 0 saves all iterations; s > 0 saves the last s iterations (i.e. the last s populations).
`initial.sd`	if init is a vector, this is the standard deviation used to seed the initial population based on perturbing init.
`cflags`	optional compiler flags to pass when compiling the GPU kernel.
`seed`	random number generator seed passed through to the sampler.
`device`	GPU device number
`backend`	execution backend. `"auto"` uses CUDA when `nvcc` is available and otherwise uses the CPU backend. `"cuda"` requires a CUDA backend. `"cpu"` uses the built-in C backend.
`nthreads`	number of OpenMP threads for the CPU backend. Ignored when OpenMP support is not available or when using the CUDA backend.
`verbose`	logical; if `TRUE`, print per-iteration acceptance-rate progress. Defaults to `FALSE`.

Details

INCA refers to interchain adaptive Markov chain Monte Carlo. In the strategy proposed by Craiu, Rosenthal and Yang (2009), several chains are run in parallel, usually from overdispersed starting values. After burn-in, the proposal kernels are adapted using information pooled from all chains. This implementation uses the global covariance version of that idea: each chain uses the same Gaussian random-walk proposal covariance, and that covariance is learned from the pooled history

of all particles. It does not implement the regional adaptive MCMC extension from the same paper. The selected target kernel evaluates an unnormalised density `f(theta)`, so the sampler targets `pi(theta)` proportional to `f(theta)`.

Random-walk Metropolis. Given a current state `theta`, RWM proposes `theta_star = theta + xi`, where `xi ~ N_D(0, Sigma_P)`. Since the Gaussian random-walk proposal is symmetric, the acceptance probability is `a(theta, theta_star) = min(1, f(theta_star) / f(theta))`.

Progress output (acceptance rate). If `verbose = TRUE`, the sampler prints a progress line of the form 'Executing iteration t of T (acceptance rate: ...)'. The displayed acceptance rate at iteration t is the fraction of the K proposals accepted at that iteration, not a cumulative acceptance rate.

Global (history-pooled) covariance learning. Let theta_t^(k) denote the state of particle k at iteration t. INCA maintains running sums over all particles and past iterations. With `n_t = K * t`, the pooled mean and population covariance are computed as:

```
theta_bar_t = S_t / n_t
Sigma_hat_t = Q_t / n_t - theta_bar_t theta_bar_t^T
```

With $s_D = 2.38^2/D$, the proposal covariance is $\Sigma_P = s_D \widehat{\Sigma}_t$ and is refreshed every `covit` iterations. The initial proposal covariance is $s_D I_D$. Cholesky factorisation is guarded by a small diagonal jitter if needed for numerical stability in single precision.

Cholesky factorisation. Gaussian proposal increments are generated via a Cholesky factor L with `Sigma_P = L L^T`. The implementation then draws `z ~ N_D(0, I_D)` and sets `xi = z L^T`. The CUDA backend uses NVIDIA cuSOLVER (`cusolverDnSpotrf`), cuRAND, and cuBLAS (`cublasStrmm`) for this step. The CPU backend computes the Cholesky factorisation and proposals in C. Random numbers are generated serially with R's RNG before the OpenMP particle-update loop, so R's global RNG state is not accessed from multiple threads.

The meaning and required layout of `GPUkernel`, `X` and `Xi` depends on the target-density kernel being used. See `cmcmc` for the currently documented kernels (including `"MVNorm"`, `"Logistic"`, `"BinomialGLM"`, `"PoissonGLM"`, `"GaussianGLM"`, `"GammaGLM"`, `"HierBetaBinomJoint"`, and `"NneWayNormalNC"`) and examples of how to construct `X` and `Xi`.

Target kernels and data layout. The `GPUkernel`, `X`, and `Xi` conventions are the same as for `cmcmc`. In particular, this package treats `X` as a flat numeric vector (copied to the GPU as single-precision floats) and `Xi` as a flat integer vector (copied as 32-bit integers). Currently documented kernels include `"MVNorm"`, `"Logistic"`, `"BinomialGLM"`, `"PoissonGLM"`, `"GaussianGLM"`, `"GammaGLM"`, `"HierBetaBinomJoint"`, and `"NneWayNormalNC"`; see `cmcmc` for the full per-kernel layouts and R code snippets. With `backend = "cpu"`, the package uses C implementations of the built-in kernels. User-provided CPU kernels can be compiled with `load_target_kernel_cpu` and then passed as `GPUkernel`. User-provided CUDA kernels are not used by the CPU backend.

Value

A numeric matrix with `2 + D` columns, where `D` is the parameter dimension (`ncol(init)`). The first two columns are metadata:

- column 1: `iter` (iteration index, 1-based; for `saved_iterations=s>0`, this will typically run from `it-s+1` to `it`),
- column 2: `particle` (particle/chain index within the population, 1..K where K=k),
- columns 3..(2+D): the state vector `theta`.

Let N denote the number of saved iterations. Then the number of rows is `K * N`. If `saved_iterations = 0` (save all), then `N = it`. If `saved_iterations = s > 0`, then `N = min(s, it)`.

References

Craiu, R. V., Rosenthal, J. S. and Yang, C. (2009). "Learn From Thy Neighbor: Parallel-Chain and Regional Adaptive MCMC." *Journal of the American Statistical Association*, 104(488), 1454–1466. doi:10.1198/jasa.2009.tm08393.

See Also

`cmcmc`, `load_target_kernel`, `load_target_kernel_cpu`, `cmcmcopt`

Examples

```
# Small multivariate normal target. This example uses the CPU backend so it
# can run on machines without CUDA.
D <- 2
K <- 100
set.seed(1)
mu <- c(0.25, -0.5)
Sigma <- matrix(c(1.0, 0.4, 0.4, 1.0), nrow = D)

# MVNorm kernel expects X = c(mu, SigmaInv) with SigmaInv in column-major order.
X <- c(mu, as.numeric(solve(Sigma)))
Xi <- integer(0)
init <- matrix(rnorm(K * D), nrow = K, ncol = D)

samples <- CMCMC::inca(
  init = init,
  GPUkernel = "MVNorm",
  it = 50,
  covit = 5,
  k = K,
  X = X,
  Xi = Xi,
  saved_iterations = 1,
  seed = 123,
  backend = "cpu"
)

# Posterior sample is the last population (K draws in D dimensions).
dim(samples)
colMeans(samples[, 3:(2 + D), drop = FALSE])
```

`load_target_kernel` *Install a user-provided CUDA target kernel*

Description

Copies a user `.cu` file into CMCMC's user-writable CUDA kernel cache so it can be compiled and used via `cmcmc(..., GPUkernel = ...)` or `inca(..., GPUkernel = ...)`. The kernel cache defaults to `file.path(tools::R_user_dir("CMCMC", "cache"), "kernels")`; set the environment variable `CMCMC_KERNEL_CACHE` to use a different writable kernel-cache root.

Usage

```
load_target_kernel(cu_path, name = basename(cu_path))
```

Arguments

`cu_path`	path to a `.cu` file defining `__device__ float logdens(...)`.
`name`	optional cached filename (defaults to basename of `cu_path`).

Details

The kernel must define:

```
__device__ float logdens(const float *theta, const float *Xf, const int *Xi);
```

The sampler calls `logdens()` for each chain state and expects the return value to be an *unnormalized log target density* for the current `theta`.

The user decides the layout of `X` and `Xi`. The sampler only copies these vectors to `logdens()`. A useful convention is to put continuous quantities, such as design matrices, numeric observations, and prior hyperparameters, in `X`, and integer quantities, such as sample sizes, group counts, indices, and count data, in `Xi`. The same layout must be used inside `logdens()`.

Kernel contract

- `theta` is a contiguous vector of length `DIM` (compile-time macro).
- `Xf` is the numeric vector passed from R through `X`, converted to `float *` on the CUDA side.
- `Xi` is the integer vector passed from R through `Xi`, converted to `int *` on the CUDA side.
- Return a finite log density when `theta` is valid, and `-INFINITY` for out-of-support/invalid states.
- Do not return `NaN`; this can break Metropolis acceptance logic.
- The function should be deterministic for fixed inputs (no RNG inside `logdens`).

Compile-time macros available in your kernel

- `DIM`: parameter dimension (`ncol(init)` at runtime).
- `K`: population size (`k` argument).

Simple kernel example

This example defines the standard multivariate normal target `pi(theta)` proportional to `exp(-0.5 * sum(theta_j^2))`, i.e. `theta ~ N(0, I_DIM)`. The returned value `-0.5 * sum(theta^2)` is the log density up to an additive constant; the normalizing term `-DIM / 2 * log(2 * pi)` is intentionally omitted, which is valid for Metropolis–Hastings because constants cancel in acceptance ratios. In this minimal example, `Xf` and `Xi` are unused, so you can call the sampler with `X = numeric()` and `Xi = integer()`.

```
#include <math.h>

__device__ float logdens(const float *theta, const float *Xf, const int *Xi) {
  (void)Xf;  // unused in this minimal example
  (void)Xi;  // unused in this minimal example

  // Standard normal target on R^DIM (up to additive constant)
  float q = 0.0f;
  for (int j = 0; j < DIM; ++j) q += theta[j] * theta[j];
  return -0.5f * q;
}
```

Value

The kernel name you can pass as `GPUkernel` (the filename).

See Also

load_target_kernel_cpu, cmcmc, inca

Examples

```
## Not run:
cu <- tempfile(fileext = ".cu")
writeLines(c(
  "#include <math.h>",
  "__device__ float logdens(const float *theta, const float *Xf, const int *Xi) {",
  "  (void)Xf; (void)Xi;",
  "  float q = 0.0f;",
  "  for (int j = 0; j < DIM; ++j) q += theta[j] * theta[j];",
  "  return -0.5f * q;",
  "}"
), cu)

kname <- load_target_kernel(cu, name = "DiagNormal.cu")
# This kernel has no data, so X and Xi are empty.
samples <- cmcmc(init = matrix(rnorm(200), nrow = 100), GPUkernel = kname,
                 it = 100, covit = 10, k = 100,
                 X = numeric(), Xi = integer(), backend = "cuda")

## End(Not run)
```

---

load_target_kernel_cpu

Compile a user-provided C target kernel for the CPU backend

Description

Compiles a user `.c` file into a shared library that can be used with `cmcmc(..., backend = "cpu", GPUkernel = ...)` or `inca(..., backend = "cpu", GPUkernel = ...)`. The shared library is built under `tempdir()`, so the returned kernel object is intended for use in the current R session. Re-run `load_target_kernel_cpu()` in a new session before using the kernel again.

Usage

```
load_target_kernel_cpu(
  c_path,
  name = tools::file_path_sans_ext(basename(c_path)),
  cflags = "",
  libs = ""
)
```

Arguments

`c_path`	path to a `.c` file defining `logdens`.
`name`	optional kernel name used for the compiled shared library.
`cflags`	optional compiler flags passed through `PKG_CFLAGS`.
`libs`	optional linker flags passed through `PKG_LIBS`.

Details

The C file must define a function named `logdens` with this signature:

```
double logdens(const double *theta, int dim,
               const double *X, int lenX,
               const int *Xi, int lenXi);
```

The sampler calls `logdens()` for each proposed chain state and expects an unnormalized log target density. Return `-INFINITY` for states outside the support and avoid returning `NaN`.

The user decides the layout of `X` and `Xi`. The sampler only passes these vectors through to `logdens()`. A useful convention is to put continuous quantities, such as design matrices, numeric observations, and prior hyperparameters, in `X`, and integer quantities, such as sample sizes, group counts, indices, and count data, in `Xi`. The same layout must be used inside `logdens()`.

Simple C kernel example

```
#include <math.h>

double logdens(const double *theta, int dim,
               const double *X, int lenX,
               const int *Xi, int lenXi) {
  (void)X; (void)lenX; (void)Xi; (void)lenXi;
  double q = 0.0;
  for (int j = 0; j < dim; ++j) q += theta[j] * theta[j];
  return -0.5 * q;
}
```

Value

A cmcmc_cpu_kernel object to pass as GPUkernel.

See Also

load_target_kernel, cmcmc, inca

Examples

```
## Not run:
cfile <- tempfile(fileext = ".c")
writeLines(c(
  "#include <math.h>",
  "#ifdef _WIN32",
  "__declspec(dllexport)",
  "#endif",
  "double logdens(const double *theta, int dim,",
  "               const double *X, int lenX,",
  "               const int *Xi, int lenXi) {",
  "  (void)X; (void)lenX; (void)Xi; (void)lenXi;",
  "  double q = 0.0;",
  "  for (int j = 0; j < dim; ++j) q += theta[j] * theta[j];",
  "  return -0.5 * q;",
  "}"
), cfile)

kname <- load_target_kernel_cpu(cfile, name = "DiagNormal")
# This kernel has no data, so X and Xi are empty.
samples <- cmcmc(init = matrix(rnorm(200), nrow = 100), GPUkernel = kname,
                 it = 100, covit = 10, k = 100,
                 X = numeric(), Xi = integer(), backend = "cpu")

## End(Not run)
```

Index