Supplementary information for "Bayesian cross-validation by parallel Markov chain Monte Carlo"

Alex Cooper[1*], Aki Vehtari[2], Catherine Forbes[1], Dan Simpson, Lauren Kennedy[1,3]

[1*]Department of Econometrics and Business Statistics, Monash University, Australia.
[2]Department of Computer Science, Aalto University, Finland.
[3]School of Computer and Mathematical Sciences, University of Adelaide, Australia.

*Corresponding author(s). E-mail(s): alexander.cooper@monash.edu;
Contributing authors: Aki.Vehtari@aalto.fi;
catherine.forbes@monash.edu; dpsimpson@gmail.com;
lauren.a.kennedy@adelaide.edu.au;

Contents

Appendix A Online parallel CV algorithms

Scoring rules that can be represented as functions of ergodic averages of the chains can also be implemented with a online estimator. Specifically, we require that we can express

$$\widehat{S}\left(p\left(\tilde{y}\,|\,y\right),\tilde{y}\right) \equiv \phi\left(\hat{\mu},\hat{\Sigma}\right) \tag{A1}$$

where $\phi : \mathbb{R}^d \times \mathbb{R}^{d\times d} \to \mathbb{R}$ and $\xi : \Theta \to \mathbb{R}^d$ are functions, and

$$\hat{\mu} = \frac{1}{S}\sum_{s=1}^{S}\xi\left(\theta^{(s)}\right) \quad \text{and} \quad \hat{\Sigma} = \frac{1}{S-1}\sum_{s=1}^{S}\left[\xi\left(\theta^{(s)}\right)-\hat{\mu}\right]\left[\xi\left(\theta^{(s)}\right)-\hat{\mu}\right]^{\top}. \tag{A2}$$

Scoring rules of this form include LogS, DSS, and HS (Table A1). Local scoring rules require only $\hat{\mu}$.

More general scoring rules, including the popular continuous ranked probability score (CRPS; Gneiting and Raftery, 2007) cannot easily be implemented with online algorithms as they typically require a full set of draws to be saved to the accelerator during inference for post processing. In addition, a need for constant memory precludes the use of many popular diagnostics, such as trace plots.

A.1 Online variance estimators

Estimating $\widehat{\eta}$ and MCMC diagnostics requires estimates for means and variances of sequences of values, computed without reference the full history of draws produced during inference. Computing the mean and variance of a sequence $x_1, \ldots, x_N$ without retaining individual values can be achieved using the method of Welford (1962).

Consider first the univariate case. We sequentially accumulate the sum $x_1 + x_2 + \ldots$ in the scalar variable A_x and sum of squares $x_1^2 + x_2^2 + \ldots$ in A_{x^2}. Then the mean $\bar{x} = A_x/N$ and

$$\text{var}(x) = \frac{1}{N-1}\sum_{i=1}^{N}\left(x_i-\bar{x}\right)^2 = \frac{1}{N-1}\left(A_{x^2}-\frac{A_x^2}{N}\right). \tag{A3}$$

In the vector case, we accumulate the sum in A_x and sum of outer squares in $A_{xx^\top}$. We will also 'center' these estimates as follows. Let $C \in \mathbb{R}^d$ be an estimate of the mean $\bar{x}$, where $d = \dim(\bar{x})$. We will accumulate the sequence of statistics $A_x^{(i)}$ and $A_{xx^\top}^{(i)}$ by each iteration evaluating the recursions

$$A_x^{(i)} = A_x^{(i-1)} + \left(x_i - C\right), \qquad A_{xx^\top}^{(i)} = A_{xx^\top}^{(i-1)} + \left(x_i - C\right)\left(x_i - C\right)^\top, \tag{A4}$$

with $A_x^{(0)} = A_{xx^\top}^{(0)} = 0$. The role of the centering constant C is to ensure that the accumulated values do not grow too large, leading to numerical overflow. We only need to allocate memory for a single d-vector $A_x^{(i)}$ and $d \times d$ matrix $A_{xx^\top}^{(i)}$. The mean

Objective	**Function** ϕ $\left(\hat{\mu}, \hat{\Sigma}\right) \mapsto$	**Function** ξ $\theta^{(s)} \mapsto$
LogS	$\log\left(\hat{\mu}\right)$	$p\left(\tilde{y} \mid \theta^{(s)}\right)$
HS	$-2\hat{\mu}_1 + \hat{\mu}_2^2$	$\left[\frac{\partial^2 \log p\left(\tilde{y}_i \mid \theta^{(s)}\right)}{\partial \tilde{y}_i^2} + \left(\frac{\partial \log p\left(\tilde{y}_i \mid \theta^{(s)}\right)}{\partial \tilde{y}_i}\right)^2\right]$ $\frac{\partial \log p\left(\tilde{y}_i \mid \theta^{(s)}\right)}{\partial \tilde{y}_i}$
DSS	$-\log\left\lvert\hat{\Sigma}\right\rvert$ $-\left(y-\hat{\mu}\right)^\top \hat{\Sigma}^{-1}\left(y-\hat{\mu}\right)$	indep. draw from $p\left(\tilde{y} \mid \theta^{(s)}\right)$

Table A1 Functions for constructing online estimators of three positively-oriented scoring rules: logarithmic score (LogS), Hyvärinen score (HS), and Dawid-Sebastini score (DSS). The estimated score $\phi : \mathbb{R}^d \times \mathbb{R}^{d\times d} \to \mathbb{R}$ is a function of $\hat{\mu}$ and $\hat{\Sigma}$, respectively the sample mean and variance of $\left\{\xi\left(\theta^{(s)}\right) : s = 1, \ldots, S\right\}$, where $\xi : \Theta \to \mathbb{R}^d$ is a function of individual MCMC parameter draws. For simplicity, this table drops CV fold indexes from the notation.

is given by $\bar{x} = A_x^{(N)}/N + C$, and the covariance can be computed as

$$\operatorname{cov}(x) = \frac{1}{N-1}\sum_{j=1}^{N}\left(x_j - C + C - \bar{x}\right)\left(x_j - C + C - \bar{x}\right)^\top \tag{A5}$$

$$= \frac{1}{N-1}\left[A_{xx^\top}^{(N)} - \frac{1}{N}A_x^{(N)}\left(A_x^{(N)}\right)^\top\right]. \tag{A6}$$

To compute the covariance of batch means of size $a \in \mathbb{N}$, replace x_i with the jth batch mean $\bar{x}_j$ and evaluate the recursions (A4) every a iterations.

To stabilize these calculations for small values like probability densities, we store values in logarithms to avoid numerical underflow (e.g. $U_x = \log A_x$). Increments of the form (A4) should use the numerically stable `logsumexp` function, available on most platforms. The need to take care to ensure numerical stability is especially acute on computing accelerators where lower-precision arithmetic (e.g. 32-bit or even 16-bit floats) is often preferred for increased performance.

A.2 Logarithmic score (LogS)

Algorithm A2 details a online PCV sampler for a single model. This sampler can serve as a building block for (say) conducting pairwise model assessment. A sample python implementation can be found in Appendix E2.

The sampler operates on all folds and chains in parallel. Within each chain, the sampler first runs a warmup loop (see Section 3). In the main inference loop is divided into a nested hierarchy of MCMC batches and shuffle blocks. The arrays U_x and U_{x^2} accumulate the log sum of predictive densities and corresponding sum of squares for each fold and chain, respectively. Similarly, V_x and V_{x^2} accumulate the same for batch means, with one increment per batch loop. Each MCMC step, the arrays Y_x and Y_{x^2} accumulate the sum of centered log densities and squared log densities, respectively,

which are required to compute $\widehat{\mathrm{R}}$. Y_x and Y_{x^2} are $K \times L \times D$ arrays, where the third dimension are the D 'shuffle blocks' required to compute the $\widehat{R}_{\max}$ benchmark.

After sampling, the final loop reshuffles Y_x and Y_{x^2} to construct the samples for the $\widehat{R}_{\max}$ benchmark.

The following functions are referenced in Algorithm A2. `Rhat` computes $\widehat{R}$ from Welford accumulators stored in logs. Note that full-chain accumulators are assumed here, so the Y variables referenced in Algorithm A2 must first be summed over the shuffle block dimension: $Y_x^c = \sum_{d=1}^{D} Y_x^{(\cdot,d)}$ and $Y_{x^2}^c = \sum_{d=1}^{D} Y_{x^2}^{(\cdot,d)}$.

$$\texttt{Rhat}\left(Y_x^c, Y_{x^2}^c\right) = \sqrt{\frac{N-1}{N} + \frac{B}{NW}} \tag{A7}$$

where we have defined

$$W = \frac{1}{L\left(N-1\right)} \sum_{\ell=1}^{L} \left[Y_{x^2}^c - \frac{1}{N} \left(Y_x^c\right)^2 \right] \quad \text{and} \quad B = \frac{N}{L-1} \sum_{\ell=1}^{L} \left[\frac{Y_x^{c(\ell)}}{N} - \frac{1}{L} \sum_{\ell'=1}^{L} \frac{Y_x^{c(\ell')}}{N} \right]^2 \tag{A8}$$

The functions `MCSE` and `ESS` compute the Monte Carlo standard error an effective sample size, respectively:

$$\texttt{MCSE}\left(V_x, V_{x^2}\right) = \sqrt{\sigma_{\hat{\eta}}^2 / LN} \qquad \text{and} \qquad \texttt{ESS}\left(U_x, U_{x^2}, V_x, V_{x^2}\right) = LN s_{\hat{\eta}}^2 / \sigma_{\hat{\eta}}^2 \tag{A9}$$

where we have defined

$$\hat{\sigma}_{\hat{\eta}}^2 = \sum_{k=1}^{K} \frac{\hat{\sigma}_{\hat{f}_k}^2}{\left(\hat{f}_k\right)^2} \qquad \hat{s}_{\hat{\eta}}^2 = \sum_{k=1}^{K} \frac{\hat{s}_{\hat{f}_k}^2}{\left(\hat{f}_k\right)^2} \tag{A10}$$

$$\hat{\sigma}_{\hat{f}_k}^2 = \frac{1}{NL} \frac{b}{LN/b - 1} \sum_{\ell=1}^{L} \left[\exp\left(V_{x^2}^{(k,\ell)}\right) - \frac{1}{N} \exp\left(2V_x^{(k,\ell)}\right) \right] \tag{A11}$$

$$\hat{s}_{\hat{f}_k}^2 = \frac{1}{NL} \frac{1}{LN - 1} \sum_{\ell=1}^{L} \left[\exp\left(U_{x^2}^{(k,\ell)}\right) - \frac{1}{N} \exp\left(2U_x^{(k,\ell)}\right) \right] \tag{A12}$$

$$\hat{f}_k = \frac{1}{NL} \sum_{\ell=1}^{L} \exp\left(U_x^{(k,\ell)}\right) \tag{A13}$$

A.3 Hyvärinen score (HS)

For a predictive density q, HS is defined as

$$\mathrm{HS}\left(q, x\right) = 2\Delta \log q\left(x\right) + \left\| \nabla \log q\left(x\right) \right\|^2, \tag{A14}$$

for ∇ the gradient and Δ the Laplacian operator. The HS is proper and key local (Dawid and Musio, 2015). HS can be computed comparatively efficiently because it is

Algorithm A2 Single-model online parallel CV sampler for LogS. Superscripts index array elements; an index of "$\cdot$" locates the vector ranged by that coordinate. By convention $\exp(-\infty) := 0$. See Section A2 for function definitions.

Input: Posterior draws $\left\{\theta_1^{(\mathrm{fd})}, \ldots, \theta_{N_{\mathrm{fd}}}^{(\mathrm{fd})}\right\}$, MCMC kernels $\{\mathcal{T}_k(\theta, \cdot)\}_{k=1}^K$, log predictive densities $\{\log p(\tilde{y}_k \mid \theta)\}_{k=1}^K$, folds K, chains L, warm-up steps N_{wu}, blocks D, batches per block G, batch size b

Output: $\widehat{S}$, $\widehat{R}$, $\widehat{R}_{\max}$, $\widehat{R}_{\max\ \mathrm{rep}}$, $\widehat{ESS}$, $\widehat{MSCE}$.

Initialize:

$K \times L$ arrays V_x, V_{x^2}, U_x, and U_{x^2} with initial entries $-\infty$; $K \times L \times G$ arrays Y with initial entries 0; $K \times R$ array $\widehat{R}_{\max\ \mathrm{rep}}$; $K \times L \times R$ arrays Y'_x, Y''_x with initial entries 0; and K-vectors C and $\widehat{R}$ with initial entries 0

for $k \in 1, \ldots, K$ **do** in parallel: ▷ fold loop

  for $\ell \in 1, \ldots, L$ **do** in parallel: ▷ chain loop

    $\theta^{(k,\ell)} \leftarrow$ draw from $\left\{\theta_1^{(\mathrm{fd})}, \ldots, \theta_{N_{\mathrm{fd}}}^{(\mathrm{fd})}\right\}$

    for $n \in 1, \ldots, N_{\mathrm{wu}}$ **do** sequentially ▷ warm-up sampling loop

      $\theta^{(k,\ell)} \leftarrow$ draw from $\mathcal{T}_k\left(\theta^{(k,\ell)}, \cdot\right)$

      $C^{(k)} \leftarrow C^{(k)} + \left[\log p\left(\tilde{y}_k \mid \theta^{(k,\ell)}\right)\right] / (LN_{\mathrm{wu}})$

    end for

    for $d \in 1, \ldots, D$ **do** sequentially ▷ block loop

      for $n \in 1, \ldots, G$ **do** sequentially ▷ batch loop

        ${Z_x}^{(k,\ell)} \leftarrow -\infty$

        for $h \in 1, \ldots, b$ **do** sequentially ▷ sampling loop

          $\theta^{(k,\ell)} \leftarrow$ draw from $\mathcal{T}_k\left(\theta^{(k,\ell)}, \cdot\right)$

          $\hat{s}^{(k,\ell)} \leftarrow \log p\left(\tilde{y}_k \mid \theta^{(k,\ell)}\right)$

          $Z_x^{(k,\ell)} \leftarrow \log\left[\exp Z_x^{(k,\ell)} + \exp \hat{s}^{(k,\ell)}\right]$

          $U_{x^2}^{(k,\ell)} \leftarrow \log\left[\exp U_{x^2}^{(k,\ell)} + \exp\left(2\hat{s}^{(k,\ell)}\right)\right]$

          $Y_x^{(k,\ell,d)} \leftarrow Y_x^{(k,\ell,d)} + \hat{s}^{(k,\ell)} - C^{(k)}$

          $Y_{x^2}^{(k,\ell,d)} \leftarrow Y_{x^2}^{(k,\ell,d)} + \left[\hat{s}^{(k,\ell)} - C^{(k)}\right]^2$

        end for

        $U_x^{(k,\ell)} \leftarrow \log\left[\exp U_x^{(k,\ell)} + \exp Z_x^{(k,\ell)}\right]$

        $V_x^{(k,\ell)} \leftarrow \log\left[\exp V_x^{(k,\ell)} + \exp\left(Z_x^{(k,\ell)} - \log H\right)\right]$

        $V_{x^2}^{(k,\ell)} \leftarrow \log\left[\exp V_{x^2}^{(k,\ell)} + \exp\left(2Z_x^{(k,\ell)} - 2\log H\right)\right]$

      end for

    end for

  end for

  for $r \in 1, \ldots, R$ **do** in parallel: ▷ shuffle draw loop

    for $\ell \leftarrow 1, \ldots, L$ **do** in parallel: ▷ chain loop

      for $d \leftarrow 1, \ldots, D$ **do** in parallel: ▷ block loop

        $\ell' \leftarrow$ draw from {1,...,L} uniformly with replacement

        ${Y'}_x^{(k,\ell,r)} \leftarrow {Y'}_x^{(k,\ell,r)} + Y_x^{(k,\ell',d)}$

        ${Y'}_{x^2}^{(k,\ell,r)} \leftarrow {Y'}_{x^2}^{(k,\ell,r)} + Y_{x^2}^{(k,\ell',d)}$

      end for

    end for

    $\widehat{R}_{\max\ \mathrm{rep}}^{(k,r)} \leftarrow \texttt{Rhat}\left({Y'}_x^{(k,\cdot,r)}, {Y'}_{x^2}^{(k,\cdot,r)}\right)$

  end for

  $\widehat{R}_k \leftarrow \texttt{Rhat}\left(\sum_{d=1}^D Y_x^{(k,\cdot,d)}, \sum_{d=1}^D Y_{x^2}^{(k,\cdot,d)}\right)$

end for

$\widehat{R}_{\max} \leftarrow \max_k \widehat{R}_k$

$\widehat{S} \leftarrow \sum_{k=1}^K \left\{\log \sum_{\ell=1}^L \exp U_x^{(k,\ell)} - \log(LDGb)\right\}$

$\widehat{ESS} \leftarrow \texttt{ESS}\left(U_x, U_{x^2}, V_x, V_{x^2}\right)$

$\widehat{MCSE} \leftarrow \texttt{MCSE}\left(V_x, V_{x^2}\right)$

Algorithm A3 Basic online parallel CV sampler for HS. Superscripts index array elements.

Input: Full-data posterior draws $\left\{\theta_1^{(\mathrm{fd})}, \ldots, \theta_{N_{\mathrm{fd}}}^{(\mathrm{fd})}\right\}$, per-fold MCMC kernels $\{\mathcal{T}_k(\theta, \cdot)\}_{k=1}^K$, per-fold log predictive functions $\left\{\log p\left(\tilde{y} \mid \theta^{(k,\ell)}\right)\right\}_{k=1}^K$, fold count K, chains per fold L, warm-up steps N_{wu}, samples per chain N

Output: HS estimate, $\widehat{HS}$.

Initialize:

K-vector C and $K \times L$ arrays Y_1 and Y_2 with initial entries 0

for $k \in 1, \ldots, K$ **do** in parallel: ▷ fold loop

for $\ell \in 1, \ldots, L$ **do** in parallel: ▷ chain loop

$\theta^{(k,\ell)} \leftarrow$ draw from $\left\{\theta_1^{(\mathrm{fd})}, \ldots, \theta_{N_{\mathrm{fd}}}^{(\mathrm{fd})}\right\}$

for $n \in 1, \ldots, N_{\mathrm{wu}}$ **do** sequentially ▷ warm-up sampling loop

$\theta^{(k,\ell)} \leftarrow$ draw from $\mathcal{T}_k\left(\theta^{(k,\ell)}, \cdot\right)$

$$C_1^{(k)} \leftarrow C_1^{(k)} + \frac{1}{LN_{wu}} \sum_{i \in \mathsf{test}_k} \left[\frac{\partial^2 \log p\left(\tilde{y}_i \mid \theta^{(k,\ell)}\right)}{\partial \tilde{y}_i^2} + \left(\frac{\partial \log p\left(\tilde{y}_i \mid \theta^{(k,\ell)}\right)}{\partial \tilde{y}_i} \right)^2 \right]$$

$$C_2^{(k)} \leftarrow C_2^{(k)} + \frac{1}{LN_{wu}} \sum_{i \in \mathsf{test}_k} \left[\frac{\partial \log p\left(\tilde{y}_i \mid \theta^{(k,\ell)}\right)}{\partial \tilde{y}_i} \right]$$

end for

for $i \in 1, \ldots, N$ **do** sequentially ▷ sampling loop

$\theta^{(k,\ell)} \leftarrow$ draw from $\mathcal{T}_k\left(\theta^{(k,\ell)}, \cdot\right)$

$$Y_1^{(k,\ell)} \leftarrow Y_1^{(k,\ell)} + \sum_{i \in \mathsf{test}_k} \left[\frac{\partial^2 \log p\left(\tilde{y} \mid \theta^{(k,\ell)}\right)}{\partial \tilde{y}^2} + \left(\frac{\partial \log p\left(\tilde{y} \mid \theta^{(k,\ell)}\right)}{\partial \tilde{y}} \right)^2 \right] - C_1^{(k)}$$

$$Y_2^{(k,\ell)} \leftarrow Y_2^{(k,\ell)} + \sum_{i \in \mathsf{test}_k} \frac{\partial \log p\left(\tilde{y} \mid \theta^{(k,\ell)}\right)}{\partial \tilde{y}} - C_2^{(k)}$$

end for

end for

$$\widehat{HS}_k \leftarrow -\frac{2}{LN} \left(\sum_{\ell=1}^L Y_1^{(k,\ell)} \right) - 2C_1^{(k,\ell)} + \left[\frac{1}{LN} \left(\sum_{\ell=1}^L Y_2^{(k,\ell)} \right) + C_2^{(k,\ell)} \right]^2$$

end for

$\widehat{HS} \leftarrow \sum_{k=1}^K \widehat{HS}_k$

homogeneous and does not depend on the normalizing term in the predictive densities (Shao et al, 2019).

The HS can be estimated using an online estimator. We will decompose HS by fold, $\mathrm{HS}(y) = \sum_{t=1}^T \mathrm{HS}(y_t)$. Shao et al (2019) show that, where exchange of differentiation and integration are justified and observations in the test set are conditionally independent, the y_{i_k} contribution to $\mathrm{HS}(y_i)$ is given by

$$2\mathbb{E}_\vartheta \left[\frac{\partial^2 \log p(y_{i_k} \mid \vartheta)}{\partial y_{i_k}^2} + \left(\frac{\partial \log p(y_{i_k} \mid \vartheta)}{\partial y_{i_k}} \right)^2 \right] - \left(\mathbb{E}_\vartheta \left[\frac{\partial \log p(y_{i_k} \mid \vartheta)}{\partial y_{i_k}} \right] \right)^2, \quad \text{(A15)}$$

where $\vartheta \sim p(\theta \mid y_{-t})$. In our examples, we have a posterior sample $\theta^{(s)} \sim p(\theta \mid y)$, for $s = 1, \ldots, S$, so we can estimate these expectations using averages.

Algorithm A4 Basic online parallel CV sampler for DSS. Superscripts index array elements. Elements of each test set $\tilde{y}_{\mathsf{test}_k}$ are presumed conditionally independent.

Input: Full-data posterior draws $\left\{\theta_1^{(\mathrm{fd})}, \ldots, \theta_{N_{\mathrm{fd}}}^{(\mathrm{fd})}\right\}$, per-fold MCMC kernels $\{\mathcal{T}_k(\theta, \cdot)\}_{k=1}^K$, functions to draw from fold predictives $\left\{p\left(\tilde{y}_{\mathsf{test}_k} \mid \theta^{(k,\ell)}\right)\right\}_{k=1}^K$, fold count K, chains per fold L, warm-up steps N_{wu}, samples per chain N

Output: DSS estimate, $\widehat{DSS}$.

Initialize:

K-vector C and $K \times L$ arrays Y_x and $Y_{xx^\top}$ with initial entries 0

for $k \in 1, \ldots, K$ **do** in parallel: ▷ fold loop

for $\ell \in 1, \ldots, L$ **do** in parallel: ▷ chain loop

$\theta^{(k,\ell)} \leftarrow$ draw from $\left\{\theta_1^{(\mathrm{fd})}, \ldots, \theta_{N_{\mathrm{fd}}}^{(\mathrm{fd})}\right\}$

for $n \in 1, \ldots, N_{\mathrm{wu}}$ **do** sequentially ▷ warm-up sampling loop

$\theta^{(k,\ell)} \leftarrow$ draw from $\mathcal{T}_k\left(\theta^{(k,\ell)}, \cdot\right)$

$\tilde{y}_{\mathsf{test}_k}^{(k,\ell)} \leftarrow$ draw from $p\left(\tilde{y}_{\mathsf{test}_k} \mid \theta^{(k,\ell)}\right)$

$C_x^{(k)} \leftarrow C_x^{(k)} + \tilde{y}_{\mathsf{test}_k}^{(k,\ell)} / \left(LN_{wu}\right)$

end for

for $i \in 1, \ldots, N$ **do** sequentially ▷ sampling loop

$\theta^{(k,\ell)} \leftarrow$ draw from $\mathcal{T}_k\left(\theta^{(k,\ell)}, \cdot\right)$

$\tilde{y}^{(k,\ell)} \leftarrow$ draw from $p\left(\tilde{y}_{\mathsf{test}_k} \mid \theta^{(k,\ell)}\right)$

$Y_x^{(k,\ell)} \leftarrow Y_x^{(k,\ell)} + \tilde{y}^{(k,\ell)} - C_x^{(k)}$

$Y_{xx^\top}^{(k,\ell)} \leftarrow Y_{xx^\top}^{(k,\ell)} + \left(\tilde{y}^{(k,\ell)} - C_x^{(k)}\right)\left(\tilde{y}^{(k,\ell)} - C_x^{(k)}\right)^\top$

end for

end for

$\hat{\mu} \leftarrow \frac{1}{LN}\left(\sum_{\ell=1}^L Y_x^{(k,\ell)}\right) + C_x^{(k)}$

$\hat{\Sigma} \leftarrow \frac{1}{LN-1}\left[\left(\sum_{\ell=1}^L Y_{xx^\top}\right) - \frac{1}{LN}\left(\sum_{\ell=1}^L Y_x^{(k,\ell)}\right)\left(\sum_{\ell=1}^L Y_x^{(k,\ell)}\right)^\top\right]$

$\widehat{\mathrm{DSS}}_k = -\log\left|\widehat{\Sigma}_k\right| - \left(y_{\mathsf{test}_k} - \widehat{\mu}_k\right)^\top \widehat{\Sigma}_k^{-1}\left(y_{\mathsf{test}_k} - \widehat{\mu}_k\right)$

end for

$\widehat{DSS} \leftarrow \sum_{k=1}^K \widehat{\mathrm{DSS}}_k$

A.4 Dawid-Sebastini score (DSS)

The DSS is defined as,

$$\widehat{\mathrm{DSS}}_{M,k}\left(y_{\mathsf{test}_k}\right) = -\log\left|\widehat{\Sigma}_{M,k}\right| - \left(\widehat{\mu}_{M,k} - y_{\mathsf{test}_k}\right)^\top \widehat{\Sigma}_{M,k}^{-1}\left(\widehat{\mu}_{M,k} - y_{\mathsf{test}_k}\right), \tag{A16}$$

where $\widehat{\mu}_{M,k}$ and $\widehat{\Sigma}_{M,k}$ are the mean and covariance of the predictive distributions for y_{test_k} under fold k for model M. To estimate $\widehat{\mu}_{M,k}$ and $\widehat{\Sigma}_{M,k}$ with a online estimator,

accumulate one predictive draw

$$\tilde{y}_{j'}^{(s)} \sim p\left(\cdot \,|\, y_{-j'}, \theta_{M,k}^{(s)}\right) \tag{A17}$$

per fold and chain, for each MCMC iteration, and compute sample mean and variance using the Welford estimator.

Appendix B Masking for parallel evaluation

PCV requires the likelihood and scoring functions for each parallel fold to incorporate different data subsets, respectively corresponding to the train_k and test_k sets. One way GPUs efficiently scale computations is by operating on data in batches, executing computations as *single instruction multiple data* (SIMD) programs (Holbrook et al, 2021; Warne et al, 2022). This requires that each fold's likelihood function be calculated by the same program code.

Parallel fold evaluation. In our experiments we implemented CV structures using data masks (arrays of binary indicator variables) to select the appropriate data subset. For the linear regression example, the N potential likelihood ℓ_k and predictive log density s_k given β and σ^2 are, for each fold $k = 1, \ldots, K$,

$$\ell_k\left(\beta, \sigma^2\right) = \sum_{n=1}^{N} \left[I_k(n) \log \mathcal{N}\left(y_n \,|\, x_n^\top \beta, \sigma^2\right)\right], \tag{B18}$$

$$s_k\left(\beta, \sigma^2\right) = \sum_{n=1}^{N} \left[I_k(n) \log \mathcal{N}\left(y_n \,|\, x_n^\top \beta, \sigma^2\right)\right]. \tag{B19}$$

In the above $I_k(n) := I\left\{n \in \mathsf{test}_k\right\}$ denotes and indicator function for membership in test_k. The following example python implementations for a log joint density and log predictive show that these expressions can be implemented without branching logic. Importantly, these functions can be parallelized: it can be evaluated in parallel with multiple different values for the `fold_id` parameter, even in the case where the folds are of heterogeneous sizes.

```
import jax.numpy as jnp
from jax.scipy.stats import norm

y = ... # array of length N
X = ... # N*p array of covariates
fold_index = ...  # integer array of length N of fold numbers

def log_joint_density(beta, sigsq, fold_id):
    fold_mask = (fold_index != fold_id)
    log_lik_all = norm.logpdf(y, loc=X @ beta, scale=jnp.sqrt(sigsq))
    log_lik_fold = (log_lik_all * fold_mask).sum()
    return log_prior + log_lik_fold

def log_pred(beta, sigsq, fold_id):
    fold_mask = (fold_index == fold_id)
    log_lik_all = norm.logpdf(y, loc=X @ beta, scale=jnp.sqrt(sigsq))
```

```
    return (log_lik_all * fold_mask).sum()
```

Admittedly, the masking approach described above does perform unnecessary calculations. Likelihood contributions are computed for all observations, including those in the test set. Similarly, predictive score contributions are computed for observations in the training set. However, on a computing accelerator where computing power is not practically constrained, this is a small price to pay for the benefit of parallel computation.

Parallel model evaluation. A similar masking approach can evaluate different models simultaneously, so long as the overall structure of the models and parameter are similar enough. The benefit is that multiple models under consideration can be evaluated in parallel; otherwise inference needs to be considered sequentially.

Consider a comparison between nested Gaussian linear regression models, with J potential covariates. Let the binary selection vector $M \in \mathbb{B}^J$ define the required model subset, where $M_j = 1$ denotes inclusion of the ith covariate. Then use the selection vector to zero out certain data elements. The log likelihood function $\ell_{k,M}\left(\beta, \sigma^2\right)$ has the form

$$\sum_{n=1}^{N}\left[I_k\left(n\right)\log\mathcal{N}\left(y_n \mid \sum_{j=1}^{J} M_j x_{n,j}\beta_j, \sigma^2\right)\right],$$

and similarly for the score function. Again, this expression can be evaluated without branching logic.

The masking approach works well for gradient-based inference methods, such as HMC. Where an element $M_j = 0$, the corresponding β_j is simply ignored by the likelihood and predictive. This poses no problem for inference, since automatic differentiation will correctly accumulate gradients for $\nabla\ell_{k,M}\left(\beta, \sigma^2\right)$, and HMC can be applied directly. However, leaving parameter elements unused *can* pose a problem for some automatic hyper-parameter tuning algorithms that rely on properties of the covariance matrix of the chains such as MEADS (Hoffman and Sountsov, 2022).

Appendix C Extra figures

	Problem size		Samples / chain		Wall time (s)	
	chains	posteriors	warmup	sampling	warmup*	sampling
Full-data (M_A)	8	1	7,000	2,000	14.7	0.5
Full-data (M_B)	8	1	7,000	2,000	10.5	0.5
PCV (M_A vs M_B)	480	60	1,000	500	10.0	7.9

Table C2 Summary of parallel leave-case-out CV for Example 2 (rat weight). Inference was performed on a 2.0 GHz Intel Xeon Cascade Lake-SP CPU with a NVIDA T4 GPU provided by Google Colab. * Includes JAX compilation time.

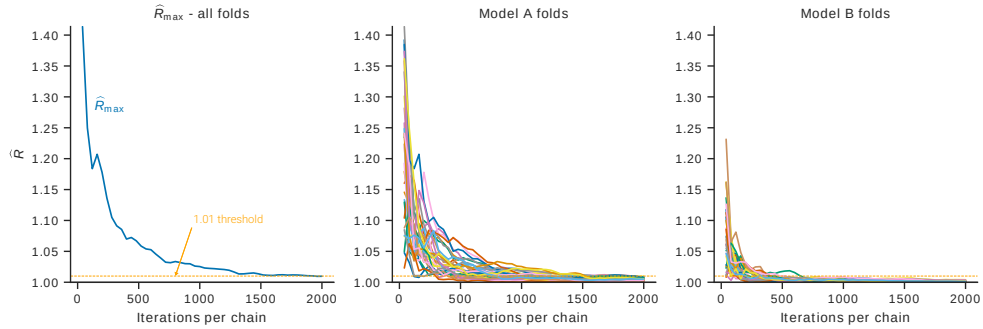

Fig. C1 Progressive $\widehat{\mathrm{R}}$ statistics for folds of Example 2 (rat weight). At least one fold $\widehat{\mathrm{R}}$ value (and hence $\widehat{R}_{\mathrm{max}}$) remains above the 1.01 threshold for at least 1,750 iterations per chain, well beyond the point where the results have stabilized.

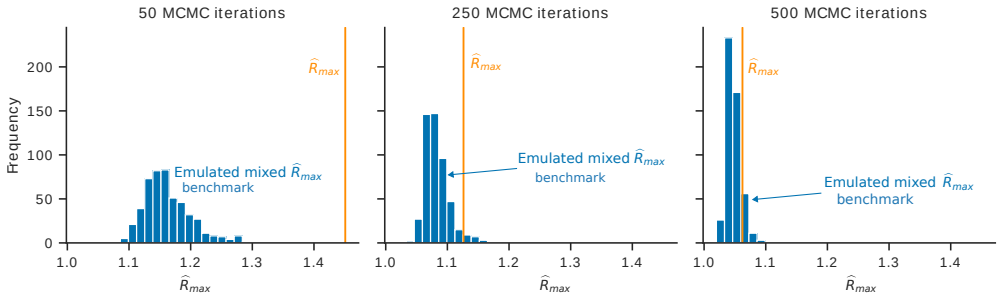

Fig. C2 $\widehat{R}_{\mathrm{max}}$ and simulated mixed threshold for three different sample sizes in Example 2 (rat weight). The plots suggest the chains have not mixed after 50 iterations, but are well mixed after 500 iterations.

	Problem size		Samples / chain		Wall time (s)	
	chains	posteriors	warmup	sampling	warmup*	sampling
Full-data (M_A)	4	1	7,000	5,000	36.8	9.4
Full-data (M_B)	4	1	7,000	5,000	28.8	7.9
PCV (M_A vs M_B)	3,088	772	2,000	2,000	47.1	45.2

Table C3 Summary of parallel CV for Example 3 (home radon). Inference was performed on a 2.0 GHz Intel Xeon Cascade Lake-SP CPU with a NVIDA A100 GPU provided by Google Colab. * Warmup includes JAX compilation.

	Problem size		Samples / chain		Wall time (s)	
	chains	posteriors	warmup	sampling	warmup*	sampling
Full-data (M_A)	4	1	10,000	2,000	19.3	2.5
Full-data (M_B)	4	1	10,000	2,000	15.4	2.0
PCV (M_A vs M_B)	3,160	790	6,000	6,000	12.9	9.8

Table C4 Summary of parallel CV for Example 4 (air passengers). Inference was performed on a 2.0 GHz Intel Xeon Skylake CPU with a NVIDA T4 GPU provided by Google Colab. *Warmup includes JAX JIT compilation step.

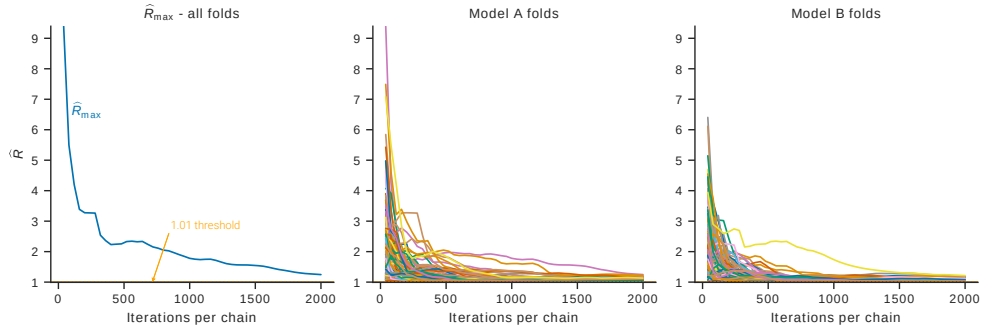

Fig. C3 Progressive $\widehat{\mathrm{R}}$ statistics for folds of Example 3 (home radon). $\widehat{R}_{\max}$ remains above 1.01 for all 4,000 iterations.

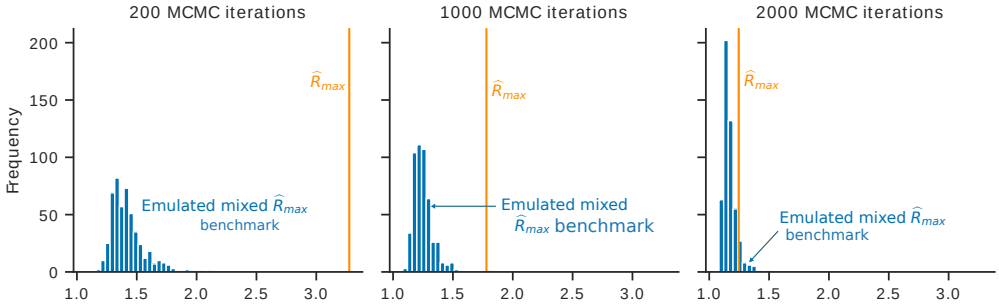

Fig. C4 $\widehat{R}_{\max}$ for three different sample sizes in Example 3 (home radon), under log loss. The benchmark is computed with 5 breaks and 500 draws. The plots suggest the results have converged by 4,000 iterations. However, notice that the $\widehat{R}_{\max}$ value is well above the 1.01 threshold suggested by Vehtari et al (2020).

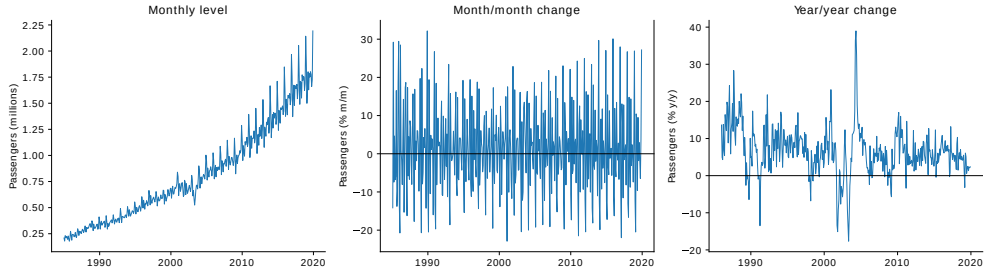

Fig. C5 International air departures from all Australian airports, 1985-2019. Source: Australian Bureau of Transport and Infrastructure Research Economic (BITRE)

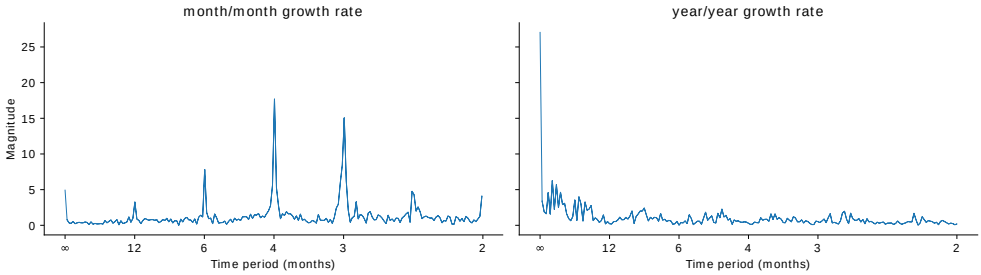

Fig. C6 Power spectrum of month/month and year/year growth rates for international air departures from all Australian airports, 1985-2019. Source: Australian Bureau of Transport and Infrastructure Research Economic (BITRE) and author's calculations.

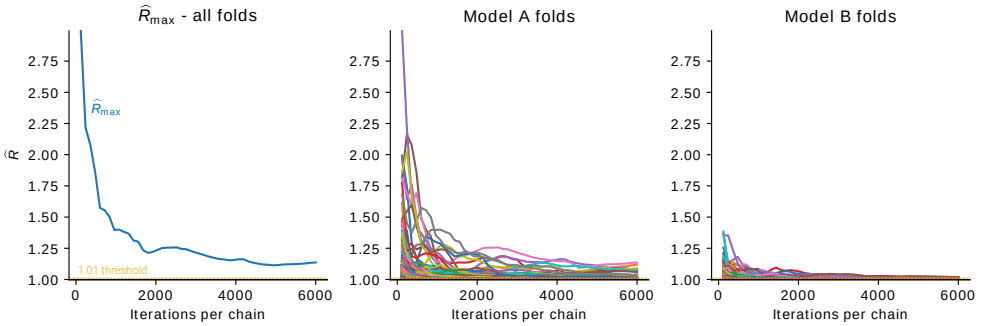

Fig. C7 Progressive $\widehat{\mathrm{R}}$ statistics for folds of Example 4 (air passengers). $\widehat{R}_{\max}$ remains well above the 1.01 threshold for all 2,000 iterations.

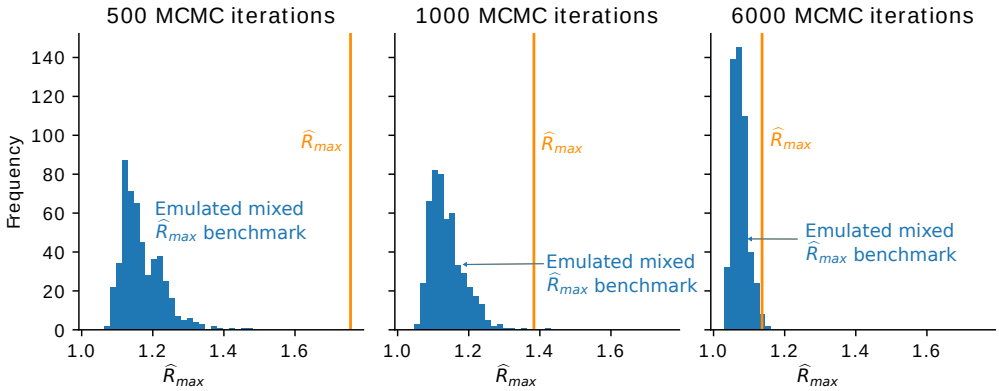

Fig. C8 $\widehat{\mathrm{R}}_{\widehat{\eta}}$ for three different sample sizes in Example 4 (air passengers). The plots suggest the results have converged at 2,000 iterations.

Appendix D Example code

This section presents minimal implementations of parallel CV sampler for LogS using python 3.10 and the JAX and BlackJAX libraries. For simplicity, the code presented here assesses predictive ability from a single model (it does not perform model selection). For a complete examples, please see https://github.com/kuperov/ParallelCV.

D.1 Non-online parallel sampler

```
import jax
import jax.numpy as jnp

def pcv_LogS_sampler(key, log_dens_fn, log_pred_fn, init_pos, K, L, N, kparam):
    """Non-online sampler for parallel cross-validation using LogS for a single model.

    Generates the predictive draws required to compute the elpd. Invoke this function
    twice to implement Algorithm 1 (once for warmup, once for sampling).

    Args:
        key: JAX PRNG key array
        log_dens_fn: log density function with signature (params, fold_id)
        log_pred_fn: log predictive density function with signature (params, fold_id)
        init_pos: K*L*p pytree of initial positions for each fold and chain
        K: number of folds
        L: number of chains
        N: chain length
        kparam: dictionary of hyperparameters for Blackjax HMC kernel

    Returns:
        Tuple: (lpred_draws, E)
    """
    def run_chain(init_pos, chain_key, fold_id, C):  # sample from a single chain
        fold_log_dens_fn = lambda params: log_dens_fn(params, fold_id)
        hmc_kernel = bj.hmc(fold_log_dens_fn, **kparam)

        def mcmc_step(carry_state, _): # a single mcmc step
            key, prev_state, E = carry_state
            step_key, carry_key = jax.random.split(key)
            state, info = hmc_kernel.step(step_key, prev_state)  # one mcmc step
            lpred_draw = log_pred_fn(state.position, fold_id)  # cond. log predictive
            E = E + info.is_divergent
            return (carry_key, state, E), lpred_draw

        init_state = hmc_kernel.init(init_pos)
        return jax.lax.scan(mcmc_step, (chain_key, init_state, 0), None, length=N)

    def run_fold(fold_key, ch_init_pos, fold_id, C): # run L chains for one fold
        sampling_fn = jax.vmap(lambda pos, key: run_chain(pos, key, fold_id, C))
        return sampling_fn(ch_init_pos, jax.random.split(fold_key, L))

    (_, _, E), lpred_draws = \
        jax.vmap(run_fold)(jax.random.split(key, K), init_pos, jnp.arange(K))
    return (lpred_draws, E)
```

D.2 Online parallel sampler

The function `pcv_LogS_sampler` samples from $K \times L$ posteriors in parallel and computes the statistics V_x, V_{x^2}, U_x, U_{x^2}, Y_x, Y_{x^2}, which are required to compute $\hat{\eta}$, $\widehat{ESS}$, and $\widehat{R}_{\max}$ (Algorithm A2). The divergence count E requires no further computation.

```
import jax
import jax.numpy as jnp

def pcv_LogS_sampler(key, log_dens_fn, log_pred_fn, init_pos, C_k, L, H, G, D, kparam):
    """Sampler for parallel cross-validation using LogS for a single model.

    Generates the statistics required to estimate ESS, MCSE, Rhat_max, and
    the Rhat_max benchmark. Space complexity is O(K*L*D), independent of
    G and H and therefore constant with respect to MCMC chain length.

    Args:
        key: JAX PRNG key array
        log_dens_fn: log density function with signature (params, fold_id)
        log_pred_fn: log predictive density function with signature (params, fold_id)
        init_pos: K*L*p pytree of initial positions for each fold and chain
        C_k: K-array of centering constants per fold
        L: number of chains
        H: MCMC draws per batch
        G: number of batches per block
        D: number of shuffle blocks
        kparam: dictionary of hyperparameters for Blackjax HMC kernel

    Returns:
        Tuple: (last_state, Ux, Ux2, Vx, Vx2, Yx, Yx2, E)
    """
    K = C_k.shape[0]

    def run_chain(init_pos, chain_key, fold_id, C):  # sample from a single chain
        fold_log_dens_fn = lambda params: log_dens_fn(params, fold_id)
        hmc_kernel = bj.hmc(fold_log_dens_fn, **kparam)

        def mcmc_step(carry_state, _): # a single mcmc step
            key, prev_state, Zx, Ux2, Yx, Yx2, E = carry_state
            step_key, carry_key = jax.random.split(key)
            params, info = hmc_kernel.step(step_key, prev_state)  # one mcmc step
            lpred = log_pred_fn(params.position, fold_id)  # cond. log predictive
            E = E + info.is_divergent
            Zx = jnp.logaddexp(Zx, lpred)  # increment accumulators
            Ux2 = jnp.logaddexp(Ux2, 2*lpred)
            Yx += lpred - C
            Yx2 += (lpred - C)**2
            return (carry_key, params, Zx, Ux2, Yx, Yx2, E), None

        def batch_step(batch_carry, _): # one batch of H mcmc steps
            key, init_state, Vx, Vx2, Ux, Ux2, Yx, Yx2, E = batch_carry
            init_carry = (key, init_state, -jnp.inf, Ux2)
            (carry_key, state, Zx, Ux2), _ = \
                jax.lax.scan(mcmc_step, init_carry, None, length=H)
            Zx_bar = Zx - jnp.log(H)  # this batch mean
            Vx = jnp.logaddexp(Vx, Zx_bar)  # increment accumulators
            Vx2 = jnp.logaddexp(Vx2, 2*Zx_bar)
            Ux = jnp.logaddexp(Ux, Zx)
            return (carry_key, state, Vx, Vx2, Ux, Ux2, Yx, Yx2, E), None
```

```
        def block_step(block_carry, _): # one block of G batches
            init_carry = block_carry + (0, 0,)
            (key, prev_state, Ux, Ux2, Vx, Vx2, Yx, Yx2, E), _ = \
                jax.lax.scan(batch_step, init_carry, None, length=G)
            return (key, prev_state, Ux, Ux2, Vx, Vx2, E), (Yx, Yx2)

        init_state = hmc_kernel.init(init_pos)
        init_carry = (chain_key, init_state, -jnp.inf, -jnp.inf, -jnp.inf, -jnp.inf)
        return jax.lax.scan(block_step, init_carry, None, length=D)

    def run_fold(fold_key, ch_init_pos, fold_id, C): # run L chains for one fold
        sampling_fn = jax.vmap(lambda pos, key: run_chain(pos, key, fold_id, C))
        return sampling_fn(ch_init_pos, jax.random.split(fold_key, L), fold_id)

    (_, last_state, Vx, Vx2, Ux, Ux2, E), (Yx, Yx2) = \
        jax.vmap(run_fold)(jax.random.split(key, K), init_pos, jnp.arange(K), C_k)
    return (last_state, Ux, Ux2, Vx, Vx2, Yx, Yx2, E)
```

References

Dawid AP, Musio M (2015) Bayesian model selection based on proper scoring rules. Bayesian Analysis 10(2):479–499. https://doi.org/10.1214/15-BA942

Gneiting T, Raftery AE (2007) Strictly proper scoring rules, prediction, and estimation. Journal of the American Statistical Association 102(477):359–378. https://doi.org/10.1198/016214506000001437

Hoffman MD, Sountsov P (2022) Tuning-Free Generalized Hamiltonian Monte Carlo

Holbrook AJ, Lemey P, Baele G, et al (2021) Massive parallelization boosts big bayesian multidimensional scaling. Journal of Computational and Graphical Statistics 30(1):11–24. https://doi.org/10.1080/10618600.2020.1754226, URL https://doi.org/10.1080/10618600.2020.1754226, pMID: 34168419, https://doi.org/10.1080/10618600.2020.1754226

Shao S, Jacob PE, Ding J, et al (2019) Bayesian model comparison with the hyvärinen score: Computation and consistency. Journal of the American Statistical Association

Vehtari A, Gelman A, Simpson D, et al (2020) Rank-normalization, folding, and localization: An improved $\widehat{R}$ for assessing convergence of MCMC. Bayesian Analysis -1(-1):1–26. https://doi.org/10.1214/20-ba1221

Warne DJ, Sisson SA, Drovandi C (2022) Vector operations for accelerating expensive bayesian computations–a tutorial guide. Bayesian Analysis 17(2):593–622

Welford BP (1962) Note on a method for calculating corrected sums of squares and products. Technometrics 4(3):419–420