# Parallelising Block Independent Metropolis-Hastings

Jack Jewson, Beniamino Hadj-Amar, Andi Wang

2/12/15

## 1 Introduction

In general, it is difficult to parallelise Monte Carlo Markov chain methods due to the nature of Markov chains themselves; in order to move forwards we need up-to-date knowledge of the current state of the chain. This naturally leads to implementation that is serial, not parallel.

The independent Metropolis-Hastings algorithm is a special case of the standard Metropolis-Hastings algorithm, whereby the the proposed values are independent of the current state of the chain. Hence, the particular proposals for a given realisation of the chain can be generated *before* the chain has even begun running. It is this key fact that gives us hope to parallelise the independent Metropolis-Hastings algorithm.

## 2 Block Independent Metropolis-Hastings

Our main algorithm is based on the Block Independent Metropolis-Hastings (BIMH) presented in Jacob et al. (2011) [1], with slight modifications; see below. We wish to generate a Markov chain of length $T$ to target some chosen density $\pi$ and to estimate $\mathbb{E}_\pi[h(X)]$ for some suitable function $h$. For a particular proposed value $y_t \sim \mu$, we write $w_t := \pi(y_t)/\mu(y_t)$.

**Algorithm 1** BIMH

Choose block size $p$ and number of blocks $b$ so $b * p = T$.
Set $x_0$ to some arbitrary value, compute $w_0 = \pi(x_0)/\mu(x_0)$.
Set $x_{\text{start}} = x_0$, $w_{\text{start}} = w_0$.
**for** $i = 1$ to $b$ **do**
    Generate new proposals $y_1, ..., y_p \sim \mu$.
    Compute corresponding $w_1, ..., w_p$.
    **for** $k = 1$ to $p$ **do**
        Uniformly draw a new permutation $\sigma$ of $\{1, \ldots, p\}$.
        Shuffle the proposed values according to $\sigma$.
        Run $p$ steps of independent MH given $(x_{\text{start}}, w_{\text{start}})$ and the (shuffled) proposed values and corresponding weights.
        Save as $x^{(k)}_{(i-1)*p+1}, ..., x^{(k)}_{i*p}$ the resulting chain.
    **end for**
    Draw index $j$ uniformly from $\{1, \ldots, p\}$, set $x_{\text{start}} = x^{(j)}_{i*p}$, $w_{\text{start}}$ correspondingly; save $x^{(j)}_{(i-1)*p+1}, \ldots, x^{(j)}_{i*p}$ into $x^*_{(i-1)*p+1}, \ldots, x^*_{i*p}$.
**end for**

The output of Algorithm 1 is $x^*_1, \ldots, x^*_T$, a Markov chain of length $T$ targeting $\pi$. We

estimate $\mathbb{E}_\pi[h(X)]$ using all values of the chain, by

$$\hat{\tau} = \frac{1}{p * T} \sum_{k=1}^{p} \sum_{t=1}^{T} h(x_t^{(k)}).$$

This algorithm is intended for cases when the most computationally expensive step is the evaluation of the target density in step 6. Note that once we have these values, in step 10 we need only calculate ratios; no more evaluations of densities are required. The output of Algorithm 1 gives a significant reduction in variance of the estimator $\hat{\tau}$ compared to an analogous estimator produced from a standard IMH run of $T$ steps. These two are comparable since in both cases we evaluate the target density $T$ times; the additional generation of uniforms and permutations in Algorithm 1 is assumed to be orders of magnitude faster.

We can also parallelise Algorithm 1; we did so in step 6 when generating the proposals and calculating weights, and in steps 8 to 11, since the $p$ chains indexed by $k$ do not interact.

Algorithm 1 differs slightly to the BIMH described in [1] since we do not generate all proposals at the start and only save $p$ at a time, and do not store all $x$-values; we update our additive estimator over the course of the algorithm. This significantly reduces the memory requirements of the algorithm and speeds up computation time. However we have also coded the BIMH algorithm exactly as in [1]; see next section.

## 3 Package Contents

For the functions in our R package, Cauchy proposals are used to target a Gaussian mixture $0.3N(0, 1) + 0.7N(5, 1)$. The C code to sample proposals and evaluate both densities are stored in '*distributions.c*' which is called in all C functions, and these are also hard coded into the R function.

1. IMH_mixNorm: a basic Independent Metropolis-Hastings algorithm.
2. serial_BIMH_mixNorm_R: a serial version of BIMH with proposals generated at the start, written entirely in R; incredibly slow.
3. Paper_serial_BIMH_mixNorm: a serial version of BIMH with proposals generated at the start, written in C; significantly faster than 2.
4. ParallelProposal_BIMH_mixNorm: same as 3 except the proposals are generated at the start in parallel.
5. GoodCaching_serial_BIMH_mixNorm: serial version of Algorithm 1 using localy generated proposals, written in C.
6. FullParallel_BIMH_mixNorm: parallel version of Algorithm 1 using localy generated proposals, written in C.

More detailed descriptions can be found in the man folder of the package.

## 4 Examples

We present some examples to demonstrate some of the functions in this package; see Table 1. We took the function $h$ to be the identity, so we are estimating the mean. We calculated the variance by running the algorithm 100 times and looking at the variance of the mean estimates.

Table 1: Some timings and the variances of the Monte Carlo estimator of the mean.

| **Method** $p = 100$, $b = 1000$ | **System.time** | **Variance (where applicable)** |
|---|---|---|
| IMH_mixNorm | user = 0.001, elapsed = 0.002 | 0.0182 |
| Paper_serial_BIMH_mixNorm | user = 0.601, elapsed = 0.630 | 0.000647 |
| GoodCaching_serial_BIMH_mixNorm | user = 0.311, elapsed = 0.310 | |
| ParallelProposal_BIMH_mixNorm | user = 0.657, elapsed = 0.640 | |
| FullParallel_BIMH_mixNorm | user = 0.791, elapsed = 0.102 | |
| serial_BIMH_MixNorm_R | user = 77.78, elapsed = 77.82 | |

We see that using BIMH over standard IMH offers a substantial improvement of the variance. However in this case it requires a lot more time. This is because for our Cauchy/Gaussian example evaluating the densities is relatively straightforward, and the majority of the computational time is spent doing the rest of the algorithm; generating permutations and uniforms and performing accept-reject. In this case the parallelising does not offer any improvement on computational time, for the same reason. We do see also that the R version of the code is indeed incredibly slow. The GoodCaching version offers a good reduction in computational time; this is because the previous version had to store a large amount of data (the entire $p * T$ by $p$ array) which would overflow the stack, reducing efficiency of retrieving data. The GoodCaching version avoids this by generating proposals within each block, where they are used and then overwriting them when no longer needed. The fact that the user time for FullParallel is much greater than the elapsed time is a sign that the parallelisation is working (although not the most efficient).

# References

[1] Jacob, P., Robert C.P. and Smith, M.H. (2011) Using parallel computation to improve independent Metropolis-Hastings based estimation. *Journal of Computational and Graphical Statistics*, 20:3, 616-635.