Debugging MCMC Code

Charles J. Geyer

April 15, 2017

1 Introduction

This document discusses debugging Markov chain Monte Carlo code using the R contributed package `mcmc` (Version 0.9-5) for examples. It also documents the debugging output of the functions `mcmc` and `temper`.

Debugging MCMC code if the code is taken as a black box is basically impossible. In interesting examples, the only thing one knows about the equilibrium distribution of an MCMC sampler is what one learns from the samples. This obviously doesn't help with testing. If the sampler is buggy, then the only thing you know about the equilibrium distribution is wrong, but if you don't know it is buggy, then you don't know it is wrong. So you don't know anything. There is no way to tell whether random output has the correct distribution when you don't know anything about the distribution it is supposed to have.

The secret to debugging MCMC code lies in two principles:

- take the randomness out, and
- expose the innards.

The first slogan means consider the algorithm a deterministic function of the elementary pseudo-random numbers that are trusted (for example, the outputs of the R random number generators, which you aren't responsible for debugging and are also well tested). The second slogan means output, at least for debugging purposes, enough intermediate state so that testing is straightforward.

For a Gibbs sampler, this means outputting all of the trusted elementary pseudo-random numbers used, the state before and after each elementary Gibbs update, and which update is being done if a random scan is used. Also one needs to output the initial seeds of the pseudo-random number generator (this is true for all situations and will not be mentioned again).

For a Metropolis-Hastings sampler, this means outputting all of the trusted elementary pseudo-random numbers used, the state before and after each elementary Metropolis-Hastings update, the proposal for that update, the Hastings ratio for that update, decision (accept or reject) in that update.

For more complicated MCMC samplers, there is more "innards" to "expose" (see the discussion of the `temper` function below), but you get the idea. You can't output too much debugging information.

2 The Metrop Function

The R function `metrop` in the `mcmc` package has an argument `debug = FALSE` that when `TRUE` causes extra debugging information to be output. Let `niter` be the number of iterations `nbatch * blen * nspac`, and let `d` be the dimension of the state vector. The result of invoking `metrop` is a list. When `debug = TRUE` it has the following additional components

- `current`, an `niter` by `d` matrix of mode `"numeric"`, the state before iteration `i` is `current[i, ]`
- `proposal`, an `niter` by `d` matrix of mode `"numeric"`, the proposal for iteration `i` is `proposal[i, ]`
- `z`, an `niter` by `d` matrix of mode `"numeric"`, the vector of standard normal random variates used to generate the proposal for iteration `i` is `z[i, ]`
- `log.green`, a vector of length `niter` and mode `"numeric"`, the logarithm of the Hastings ratio for each iteration
- `u`, a vector of length `niter` and mode `"numeric"`, the Uniform$(0, 1)$ random variate compared to the Hastings ratio for each iteration or `NA` if none is needed (when the log Hastings ratio is nonnegative)
- `debug.accept`, a vector of length `niter` and mode `"logical"`, the decision for each iteration, accept the proposal (`TRUE`) or reject it (`FALSE`)

(The components `z` and `debug.accept` were added in version 0.7-3 of the `mcmc` package. Before that only the others were output.)

Two components of the list returned by the `metrop` function always (whether `debug = TRUE` or `debug = FALSE`) are also necessary for debugging. They are

- `initial.seed` the value of the variable `.Random.seed` that contains the seeds of the R random number generator system before invocation of the `metrop` function
- `final`, a vector of length `d` and mode `"numeric"`, the state after the last iteration

All of the files in the `tests` directory of the source for the package (not installed but found in the source tarball on CRAN) test the `metrop` function except those beginning `temp`, which test the `temper` function. Since these tests were written many years ago, are spread out over many files, and are not commented, we will not describe them in detail. Suffice it to say that they check every aspect of the functioning of the `metrop` function.

3 The Temper Function

The R function `temper` in the `mcmc` package has an argument `debug = FALSE` that when `TRUE` causes extra debugging information to be output. Let `niter` be the number of iterations `nbatch * blen * nspac`, let `d` be the dimension of the state vector, and let `ncomp` be the number of components of the tempering mixture. The result of invoking `temper` is a list. When `debug = TRUE` and `parallel = TRUE` it has the following additional components

- `which`, a vector of length `niter` and mode `"logical"` the type of update for each iteration, within component (`TRUE`) or swap components (`FALSE`).
- `unif.which`, a vector of length `niter` and mode `"numeric"`, the Uniform$(0, 1)$ random variate used to decide which type of update is done.
- `state`, an `niter` by `ncomp` by `d` array of mode `"numeric"`, the state before iteration `i` is `state[i, , ]`
- `proposal`, an `niter` by `d + 1` matrix of mode `"numeric"`, the proposal for iteration `i` is `proposal[i, ]` (explanation below)
- `coproposal`, an `niter` by `d + 1` matrix of mode `"numeric"`, the proposal for iteration `i` is `coproposal[i, ]` (explanation below)
- `log.hastings`, a vector of length `niter` and mode `"numeric"`, the logarithm of the Hastings ratio for each iteration
- `unif.hastings`, a vector of length `niter` and mode `"numeric"`, the Uniform$(0, 1)$ random variate compared to the Hastings ratio for each iteration or `NA` if none is needed (when the log Hastings ratio is nonnegative)
- `acceptd`, a vector of length `niter` and mode `"logical"`, the decision for each iteration, accept the proposal (`TRUE`) or reject it (`FALSE`)
- `norm`, an `niter` by `d` matrix of mode `"numeric"`, the vector of standard normal random variates used to generate the proposal for iteration `i` is `z[i, ]` unless none are needed (for swap updates) when it is `NA`
- `unif.choose`, an `niter` by 2 matrix of mode `"numeric"`, the vector of Uniform$(0, 1)$ random variates used to choose the components to update in iteration `i` is `unif.choose[i, ]`; in a swap update two are used; in a within-component update only one is used and the second is `NA`

In a within-component update, one component say `j` is chosen for update. The *coproposal* is the current value of the state for this component, which is a vector of length `d + 1`, the first component of which is `j` and the rest of which is `state[i, j, ]` if we are in iteration `i`. The *proposal* is a similar vector, the first component of which is again `j` and the rest of which is a multivariate normal random vector centered at `state[i, j, ]`. The coproposal is the current state; the proposal is the possible value (if accepted) of the state at the next time.

In a swap update, two components say `j1` and `j2` are chosen for update. Strictly, speaking the coproposal is the pair of vectors `c(j1, state[i, j1, ])` and `c(j2, state[i, j2, ])` and the proposal is these swapped, that is, the pair of vectors `c(j2, state[i, j1, ])` and `c(j1, state[i, j2, ])` if we are in iteration `i`. Since, however, there is a lot of redundant information here, the vector `c(j1, state[i, j1, ])` is output as `coproposal[i, ]` and the vector `c(j2, state[i, j2, ])` is output as `proposal[i, ]`.

When `debug = TRUE` and `parallel = FALSE` the result of invoking `temper` is a list having the following additional components

- `which`, a vector of length `niter` and mode `"logical"` the type of update for each iteration, within component (`TRUE`) or jump from one component to another (`FALSE`).
- `unif.which`, a vector of length `niter` and mode `"numeric"`, the Uniform$(0, 1)$ random variate used to decide which type of update is done.
- `state`, an `niter` by `d + 1` matrix of mode `"numeric"`, the state before iteration `i` is `state[i, ]`
- `proposal`, an `niter` by `d + 1` matrix of mode `"numeric"`, the proposal for iteration `i` is `proposal[i, ]`
- `log.hastings`, a vector of length `niter` and mode `"numeric"`, the logarithm of the Hastings ratio for each iteration
- `unif.hastings`, a vector of length `niter` and mode `"numeric"`, the Uniform$(0, 1)$ random variate compared to the Hastings ratio for each iteration or `NA` if none is needed (when the log Hastings ratio is nonnegative)
- `acceptd`, a vector of length `niter` and mode `"logical"`, the decision for each iteration, accept the proposal (`TRUE`) or reject it (`FALSE`)
- `norm`, an `niter` by `d` matrix of mode `"numeric"`, the vector of standard normal random variates used to generate the proposal for iteration `i` is `z[i, ]` unless none are needed (for jump updates) when it is `NA`
- `unif.choose`, a vector of length `niter` and mode `"numeric"`, the Uniform$(0, 1)$ random variates used to choose the component to update in iteration `i` is `unif.choose[i, ]`; in a jump update one is used; in a within-component update none is used and `NA` is output

All of the files in the `tests` directory of the source for the package (not installed but found in the source tarball on CRAN) beginning `temp` test the `temper` function. They check every aspect of the functioning of the `temper` function.

In the file `temp-par.R` in the `tests` directory, the following checks are made according to the comments in that file

1. check decision about within-component or jump/swap

2. check proposal and coproposal are actually current state or part thereof
3. check hastings ratio calculated correctly
4. check hastings rejection decided correctly
5. check acceptance carried out or not (according to decision) correctly
6. check within-component proposal
7. check swap proposal
8. check standard normal and uniform random numbers are as purported
9. check batch means
10. check acceptance rates
11. check scale vector
12. check scale matrix
13. check scale list
14. check outfun

In the file `temp-ser.R` in the `tests` directory, the all of the same checks are made according to the comments in that file except for check number 2 above, which would make no sense because there is no `coproposal` component in the serial (`parallel = FALSE`) case.