An MCMC Package for R

Charles J. Geyer

December 17, 2011

1 Introduction

This package is a simple first attempt at a sensible *general* MCMC package. It doesn't do much yet. It only does "normal random-walk" Metropolis for continuous distributions. No non-normal proposals. No Metropolis-Hastings or Metropolis-Hastings-Green. No discrete state. No dimension jumping. No simulated tempering. No perfect sampling. There's a lot left to do. Still, limited as it is, it does equilibrium distributions that no other R package does.

Its basic idea is the following. Given an R function `fred` that calculates the unnormalized density of the desired equilibrium distribution of the Markov chain, or, better yet, *log* unnormalized density, so we avoid overflow and underflow, the `metrop` function should generate a Markov chain having this stationary distribution.

The package does not do any of the following.

- **Theory.** (What R package does?) It doesn't prove the Markov chain is irreducible or ergodic or positive recurrent or Harris recurrent or geometrically ergodic or uniformly ergodic or satisfies conditions for the central limit theorem.

- **Diagnostics.** There are no non-bogus Markov chain diagnostics (except for perfect sampling). This package doesn't do any bogus diagnostics (other R packages do them).

- **Calculus.** If the putative unnormalized density specified by `fred` is not integrable, then it does not specify an equilibrium distribution. But this package doesn't check that either.

Thus the only requirement the package has to satisfy is that given a function `fred` it correctly simulates a Markov chain that actually has `fred` as its equilibrium distribution (when `fred` actually does specify some equilibrium distribution)

2 Design Issues

2.1 First Try

For a start we have a function with signature

```
metrop(lud, initial, niter, ...)
```

such that when

- `initial` is a real vector, the initial state of the Markov chain,
- `lud` is a function, the log unnormalized density of the equilibrium distribution of the Markov chain, such that
  - `lud(initial, ...)` works and produces a finite scalar value and
  - `lud(x, ...)` works for any real vector `x` having the same length as `initial` and all elements finite and and produces a scalar value that is finite or `-Inf`,

then the function produces an `niter` by `length(initial)` matrix whose rows are the iterations of the Markov chain.

2.1.1 Checks

If

```
logh <- lud(initial, ...)
```

then `is.finite(logh)` is `TRUE`.

Moreover, if `x` is any vector such that `length(x) == length(initial)` and `all(is.finite(x))` are `TRUE` and

```
logh <- lud(x, ...)
```

then

```
is.finite(logh) | (logh == -Inf)
```

is `TRUE`.

Points `x` having log unnormalized density `-Inf` have density zero (normalized or unnormalized, since a constant times zero is zero) hence cannot occur. Thus if

```
path <- metrop(fred, x, n, some, extra, arguments)
```

then

```
all(is.finite(apply(path, 1, fred, some, extra, arguments)))
```

is `TRUE`.

This is how we specify log unnormalized densities for distribution having support that is not all of Euclidean space. The value of the log unnormalized density off the support is `-Inf`.

Thus when coding a log unnormalized density, we should normally do something like

```
fred <- function(x, ...)
{
    if (! is.numeric(x))
        stop("argument x not numeric")
    if (length(x) != d)
        stop("argument x wrong length")
    if (! all(is.finite(x)))
        stop("elements of argument x not all finite")
    if (! is.in.the.support(x))
        return(-Inf)
    return(log.unnormalized.density(x))
}
```

where `d` is the dimension of the state space of the Markov chain (defined in the global environment or in the `...` arguments), `is.in.the.support(x)` returns `TRUE` if `x` is in the support of the desired equilibrium distribution and `FALSE` otherwise and `log.unnormalized.density(x)` calculates the log unnormalized density of the desired equilibrium distribution at the point `x`, which is guaranteed to be finite because `x` is in the support if the this code is executed.

Of course, you needn't actually have functions named `is.in.the.support` and `log.unnormalized.density`. The point is that you use this logic. First you check whether `x` is in the support. If not return `-Inf`. If it is, return a finite value. Do not crash. Do not return `NA`, `NaN`, or `Inf`. If you do, then `metrop` crashes, and it's your fault.

Of course, a crash is no big deal. Lots of first efforts in R crash. You just fix the problem and retry. Error messages are your friends.

2.2 Proposal

We also need to specify the proposal distribution (the preceding stuff assumed some default proposal). This can be any multivariate normal distribution on the Euclidean space of dimension `length(initial)` having mean zero. Thus it is specified by specifying its covariance matrix.

But to avoid having to check whether the specified covariance matrix actually is a covariance matrix, we make the specification an arbitrary `d` by `d` matrix, call it `scale`, where `d` is the dimension of the state space, specified by `length(initial)`, and use the proposal `x + scale %*% z`, where `x` is the current state and `z` is a standard normal random vector ("standard" meaning its covariance matrix is the identity matrix).

Thus we need to add this to the argument list of our function. It is now

```
metrop(lud, initial, niter, scale, ...)
```

The covariance matrix specified by this is, of course, `scale %*% t(scale)`. If you want the proposal to have covariance matrix `melvin`, then specifying `scale = t(chol(melvin))` will do the job. (Of course, many other specifications will also do the job.)

For convenience, we also allow `scale` to be a vector of length `d` and in this case take `scale = sally` to mean the same thing as `scale = diag(sally)`.

For convenience, we also allow `scale` to be a vector of length 1 and in this case take `scale = sally` to mean the same thing as `scale = sally * diag(d)` where `d` is still the dimension of the state space `length(initial)`.

We can use this last convenience option to give `scale` a default

```
metrop(lud, initial, niter, scale = 1, ...)
```

In order to tell what is sensible scaling, we need to return the acceptance rate (the proportion of proposals that are accepted). The only criterion known for choosing sensible scaling is to adjust so that the acceptance rate is about 20%. Of course, that recommendation was derived for a specific toy model that is not very much like what people do in real MCMC applications, so 20% is only a very rough guideline. But acceptance rate is all we know to use, so that's what we will output.

Thus the result of `metrop`, assuming we write it in R will be something like

```
return(structure(list(path = path, rate = rate),
    class = "mcmc"))
```

and if we write it in C will be whatever does the same job.

2.3 Output I

Generally we don't want `path` to be as described above. It may be way too big. We might have `d`, the dimension of the state space 10^3 or even larger and we might have `niter` 10^7 or even larger, the resulting `path` matrix would be 10^{10} doubles or 8×10^{10} bytes. Too big to fit in my computer!

Thus we facilitate subsampling and batching of the output.

2.3.1 Subsampling

If the Markov chain exhibits high autocorrelation, subsampling the chain may lose little information. (Most users way overdo the subsampling, but it's not the job of a computer program to keep users from overdoing things). Thus we add an argument `nspac` that specifies subsampling. Only every `nspac` iterate is output.

2.3.2 Batching

The method of batch means uses “batches” which are sums over consecutive blocks of output. For most purposes batching is better than subsampling. It loses no information while reducing the amount of output even more than subsampling. So we introduce arguments `nbatch` specifying the number of batches and `blen` specifying the length of the batches.

Our function now has signature

```
metrop(lud, initial, nbatch, blen = 1, nspac = 1, scale = 1,
    ...)
```

Note that the argument `niter` formerly present has vanished. The number of iterations that will now be done is `nbatch * blen * nspac`. If we accept the defaults `blen = 1` and `nspac = 1`, then `nbatch` is the same as the former argument `niter`. Otherwise, it is quite different.

2.4 Output II

The preceding section takes care of of the problem of `niter` being too big. This section deals with the dimension of the state space being too big. When the dimension of the state space is large, we generally do not want to output the whole state, but only some function of the state.

Thus we need another function (besides `lud`) to produce the output vector. Call it `outfun`. The requirements on `outfun` are

- If `is.finite(lud(x, ...))`, then `outfun(x, ...)` works (it does not crash) and produces a vector having all elements finite and always of the same length (say `k`).

`outfun` will never be called in any other situation (that is, never when `x` is not in the support of the equilibrium distribution).

Now we can describe the `path` component of the output. We’ll use a little math here. Write L for `blen` and M for `nspac`. Write x_i for the i-th iterate of the Markov chain, and write g for `outfun`. Then `path[`j`, ]` is the vector

$$\frac{1}{L} \sum_{i=1}^{L} g(x_{M[L(j-1)+i]})$$

For convenience, we also allow `outfun` to be a logical vector of length `d` or an integer vector having elements in `1:d` or in `-(1:d)` and in this case take `outfun = fred` to mean the same thing as `outfun = function(x) x[fred]`.

For convenience, we also allow `outfun` to be missing take this to mean the same thing as `outfun = function(x) x`, that is, the “outfun” is the identity function and we are back to outputting the entire state.

Our function now has signature

```
metrop(lud, initial, nbatch, blen = 1, nspac = 1, scale = 1,
    outfun, ...)
```

2.5 Restarting

It should be possible to restart the Markov chain and get exactly the same results. It should be possible to continue the Markov chain and get exactly the same results. Thus we need to save the initial and final state of the Markov chain and the initial and final state of the random number generator (the R object `.Random.seed`).

Thus the result of `metrop` now looks like

```
return(structure(list(path = path, rate = rate,
    initial = initial, final = final,
    initial.seed = iseed, final.seed = .Random.seed),
    class = "mcmc"))
```

We also need to add arguments to `metrop`. It now has signature

```
metrop(lud, initial, nbatch, blen = 1, nspac = 1, scale = 1,
    outfun, object, restart = FALSE, ...)
```

Here `object` is an R object of class `"mcmc"`, the output a previous call to `metrop`, from which we take either initial or final state and seed depending the value of `restart`.

While we are at it, it is convenient to allow any or all of the other arguments to be missing if `object` is supplied. We just take the argument from `object`. Thus we can make calls like

```
out <- metrop(fred, x, 1e3, scale = 4, blen = 3)
out.too <- metrop(object = out, nbatch = 1e4)
```

Woof! I now see (how embarrasing) after four earlier versions how to use the R class system to make this convenient. We have three functions.

```
metrop.default <- function(o, ...)
UseMethod("metrop")

metrop.mcmc <- function(o, initial, nbatch, blen = 1,
    nspac = 1, scale = 1, outfun, restart = FALSE, ...)
{
    if (missing(nbatch)) nbatch <- o$nbatch
    if (missing(blen)) blen <- o$blen
    if (missing(nspac)) nspac <- o$nspac
    if (missing(scale)) scale <- o$scale
    if (missing(outfun)) outfun <- o$outfun

    if (restart) {
        .Random.seed <- o$final.seed
        return(metrop.function(o$lud, o$final, nbatch, blen,
            nspac, scale, outfun))
    } else {
        .Random.seed <- o$initial.seed
        return(metrop.function(o$lud, o$initial, nbatch, blen,
            nspac, scale, outfun))
    }
}

metrop.function <- function(o, initial, nbatch, blen = 1,
    nspac = 1, scale = 1, outfun, restart = FALSE, ...)
{
    if (! exists(".Random.seed")) runif(1)
    initial.seed <- .Random.seed
    func1 <- function(state) o(state, ...)
    func2 <- function(state) outfun(state, ...)
    .Call("metrop", func1, initial, nbatch, blen,
        nspac, scale, func2, environment(fun = func1),
        environment(fun = func2))
}
```

Note that `restart` is ignored in `metrop.function`. We can't "restart" when we have no saved state in an `"mcmc"` object.

Note also that our `"mcmc"` objects must now store a lot more stuff (and more to come in the next section).

2.6 Testing and Debugging

It is nearly impossible to test or debug a random algorithm (any Monte Carlo) from looking at its designed (useful to the user) output. In order to do any serious testing or debugging, it is necessary to look under the hood. For the Metropolis algorithm, we need to look at the current state, the proposal, the log odds ratio, the uniform random variate (if any) used in the Metropolis rejection, and the result (accept or reject) of the Metropolis rejection.

Hence we need to add one final argument `debug = FALSE` to our functions and a lot of debugging output to the result.

In debugging a Metropolis (etc.) algorithm there is a very important principle. Debugging should use Markov chain theory! Just enlarge the state space of the Markov chain to include

- the proposal (vector),
- details of the calculation of the Metropolis-Hastings-Green ratio (for Metropolis this is just the log unnormalized density at the current state and proposal, for others it includes proposal densities) and the calculated ratio,
- the uniform random number (if any) used in the decision, and
- the decision (`TRUE` or `FALSE`) in the Metropolis rejection.

With all that it is easy to tell whether the algorithm is doing the Right Thing. Without all that, it's nearly impossible.