How to code up a general Metropolis sampler in R

Mattias Villani

The function signature

This note explains how to code up a Random Walk Metropolis (RWM) algorithm in a fairly general way so that the same function can be used to simulate from the posterior distribution for *any* model. The trick is to use:

- **function objects** as input argument to supply the posterior density
- **triple dot** (**...**) to supply arbitrary arguments (data and prior hyperparameters) to the posterior density.

  Here is the signature for the `RWMsampler` function. Your task is to fill in the function body. But first some explanation of the how to specify the `logPostFunc` function and the triple dots `...` argument.

  ```
  RWMsampler <- function(logPostFunc, initVal, nSim, nBurn, Sigma, c, ...){
    # Run the algorithm for nSim iterations starting at theta = initVal
    # using the multivariate proposal N(theta_previous_draw, c*Sigma)
    # Return the posterior draws after discarding nBurn iterations as burn-in
  }
  ```

The log posterior function object

One of the input arguments of your `RWMSampler` function should be `logPostFunc` (or some other suitable name). `logPostFunc` is a *function object* that computes the log posterior density (proportional form is enough)

$$\log p(\theta|x) \propto \log p(x|\theta) + \log p(\theta)$$

at any value of the parameter vector θ for a given dataset x. This is needed when you compute the acceptance probability of the Metropolis algorithm. **Always** code up the *log* posterior density, since logs are more stable and avoids problems with too small or large numbers (overflow). Note that the ratio of posterior densities in the Metropolis acceptance probability can be written

$$\frac{p(\theta^\star|x)}{p(\theta^{(i)}|x)} = \exp\left(\log p(\theta^\star|x) - \log p(\theta^{(i)}|x)\right)$$

This is clever since common multiplicative factors in $p(\theta^\star|x)$ and $p(\theta^{(i)}|x)$ cancel out before we evaluate the exponential function (which can otherwise overflow).

The first argument of your (log) posterior function should be `theta`, the vector of parameters for which the posterior density is evaluated. You can of course use some other name for the variable, but it must be the *first* argument of your posterior density function.

Here is an example with the log posterior function for the iid Bernoulli model with a $\theta \sim \text{Beta}(a, b)$ prior:

```r
LogPostBernBeta <- function(theta, s, f, a, b){
  logLik = s*log(theta) + f*log(1 - theta)
  logPrior = dbeta(theta, a, b, log = TRUE)
  logPost = logLik + logPrior
  return(logPost)
}
```

We can test that the function works:

```r
# Testing if the log posterior function works
s = 8
f = 2
a = 2
b = 2
logPost = LogPostBernBeta(theta = 0.3, s, f, a, b)
print(logPost)
```

```
[1] -10.11402
```

Wildcard arguments with triple dots ...

The user's posterior density is also a function of the data and prior hyperparameters and those can can be supplied to the `RWMSampler` function by using the triple dot (`...`) argument to functions. The triple dot acts like a wildcard for *any* parameters supplied by the user. This makes it possible to use your `RWMsampler` function for any problem, even when you as a programmer don't know what the user's posterior density function looks like or what kind of data and hyperparameters will be used in that particular problem.

To illustrate the use of the triple dot argument, here is a stupid function that evaluates the input function `myFunction` at `x=0.3` and then multiplies that output by the number 2. Note that `myFunction` is allowed to have any number of arguments due to the wildcard triple dot argument `...`

```r
MultiplyByTwo <- function(myFunction, ...){
    x = 0.3
    y = myFunction(x,...)
    return(2*y)
}
```

Let's try it on the log posterior for the Bernoulli model above:

```r
#Let's try if the MultiplyByTwo function works:
MultiplyByTwo(LogPostBernBeta, s, f, a, b)
```

```
[1] -20.22804
```

where the triple dots `...` in `MultiplyByTwo` catches all of the inputs `s`, `f`, `a`, and `b` .

We can apply the same `MultiplyByTwo` function to a completely different function with completely different arguments:

```r
# Define the power function
powerFunction <- function(x, power){
  return(x^power)
}
# and now apply the same MultiplyByTwo() function:
c = 3 # third power, i.e. cubic function.
MultiplyByTwo(powerFunction, c) # this returns 2 times (0.3)^3 = 0.054
```

```
[1] 0.054
```