MCMC Package Example (Version 0.9.7)

Charles J. Geyer

March 20, 2020

1 The Problem

This is an example of using the `mcmc` package in R. The problem comes from a take-home question on a (take-home) PhD qualifying exam (School of Statistics, University of Minnesota).

Simulated data for the problem are in the dataset `logit`. There are five variables in the data set, the response y and four predictors, x1, x2, x3, and x4.

A frequentist analysis for the problem is done by the following R statements

```
> library(mcmc)
> data(logit)
> out <- glm(y ~ x1 + x2 + x3 + x4, data = logit,
+     family = binomial, x = TRUE)
> summary(out)

Call:
glm(formula = y ~ x1 + x2 + x3 + x4, family = binomial, data = logit,
    x = TRUE)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-1.7461  -0.6907   0.1540   0.7041   2.1943

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)   0.6328     0.3007   2.104  0.03536 *
x1            0.7390     0.3616   2.043  0.04100 *
x2            1.1137     0.3627   3.071  0.00213 **
x3            0.4781     0.3538   1.351  0.17663
x4            0.6944     0.3989   1.741  0.08172 .
---
Signif. codes:
0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)
```

```
    Null deviance: 137.628  on 99  degrees of freedom
Residual deviance:  87.668  on 95  degrees of freedom
AIC: 97.668

Number of Fisher Scoring iterations: 6
```

But this problem isn't about that frequentist analysis, we want a Bayesian analysis. For our Bayesian analysis we assume the same data model as the frequentist, and we assume the prior distribution of the five parameters (the regression coefficients) makes them independent and identically normally distributed with mean 0 and standard deviation 2.

The log unnormalized posterior density (log likelihood plus log prior) for this model is calculated by the following R function. In order to avoid using either global variables or ... arguments, we use the function factory pattern (Wickham, 2014, Section 10.3.1; Geyer, 2018, Section 7.5, especially Subsection 7.5.4; see also Appendix A below).

```
> lupost_factory <- function(x, y) function(beta) {
+     eta <- as.numeric(x %*% beta)
+     logp <- ifelse(eta < 0, eta - log1p(exp(eta)), - log1p(exp(- eta)))
+     logq <- ifelse(eta < 0, - log1p(exp(eta)), - eta - log1p(exp(- eta)))
+     logl <- sum(logp[y == 1]) + sum(logq[y == 0])
+     return(logl - sum(beta^2) / 8)
+ }
> lupost <- lupost_factory(out$x, out$y)
```

The tricky calculation of the log likelihood avoids overflow and catastrophic cancellation in calculation of $\log(p)$ and $\log(q)$ where

$$p = \frac{\exp(\eta)}{1 + \exp(\eta)} = \frac{1}{1 + \exp(-\eta)}$$

$$q = \frac{1}{1 + \exp(\eta)} = \frac{\exp(-\eta)}{1 + \exp(-\eta)}$$

so taking logs gives

$$\log(p) = \eta - \log(1 + \exp(\eta)) = -\log(1 + \exp(-\eta))$$
$$\log(q) = -\log(1 + \exp(\eta)) = -\eta - \log(1 + \exp(-\eta))$$

To avoid overflow, we always chose the case where the argument of exp is negative. We have also avoided catastrophic cancellation when $|\eta|$ is large. If η is large and positive, then

$$p \approx 1$$
$$q \approx 0$$
$$\log(p) \approx -\exp(-\eta)$$
$$\log(q) \approx -\eta - \exp(-\eta)$$

2

and our use of the R function `log1p`, which calculates the function $x \mapsto \log(1+x)$ correctly for small x avoids all problems. The case where η is large and negative is similar.

2 Beginning MCMC

With those definitions in place, the following code runs the Metropolis algorithm to simulate the posterior.

```
> set.seed(42)     # to get reproducible results
> beta.init <- as.numeric(coefficients(out))
> out <- metrop(lupost, beta.init, 1e3)
> names(out)

 [1] "accept"       "batch"        "initial"
 [4] "final"        "accept.batch" "initial.seed"
 [7] "final.seed"   "time"         "lud"
[10] "nbatch"       "blen"         "nspac"
[13] "scale"        "debug"

> out$accept

[1] 0.008
```

The arguments to the `metrop` function used here (there are others we don't use) are

- an R function (here `lupost`) that evaluates the log unnormalized density of the desired stationary distribution of the Markov chain (here a posterior distribution). Note that (although this example does not exhibit the phenomenon) that the unnormalized density may be zero, in which case the log unnormalized density is `-Inf`.

- an initial state (here `beta.init`) of the Markov chain.

- a number of batches (here `1e3`) for the Markov chain. This combines with batch length and spacing (both 1 by default) to determine the number of iterations done.

- additional arguments (here `x` and `y`) supplied to provided functions (here `lupost`).

- there is no "burn-in" argument, although burn-in is easily accomplished, if desired (more on this below).

The output is in the component `out$batch` returned by the `metrop` function. We'll look at it presently, but first we need to adjust the proposal to get a higher acceptance rate (`out$accept`). It is generally accepted (Gelman, Roberts, and

3

Gilks, 1996) that an acceptance rate of about 20% is right, although this recommendation is based on the asymptotic analysis of a toy problem (simulating a multivariate normal distribution) for which one would never use MCMC and is very unrepresentative of difficult MCMC applications.

Geyer and Thompson (1995) came to a similar conclusion, that a 20% acceptance rate is about right, in a very different situation. But they also warned that a 20% acceptance rate could be very wrong and produced an example where a 20% acceptance rate was impossible and attempting to reduce the acceptance rate below 70% would keep the sampler from ever visiting part of the state space. So the 20% magic number must be considered like other rules of thumb we teach in intro courses (like $n > 30$ means means normal approximation is valid). We know these rules of thumb can fail. There are examples in the literature where they do fail. We keep repeating them because we want something simple to tell beginners, and they are all right for some problems.

Be that as it may, we try for 20%.

```
> out <- metrop(out, scale = 0.1)
> out$accept

[1] 0.739

> out <- metrop(out, scale = 0.3)
> out$accept

[1] 0.371

> out <- metrop(out, scale = 0.5)
> out$accept

[1] 0.148

> out <- metrop(out, scale = 0.4)
> out$accept

[1] 0.209
```

Here the first argument to each instance of the `metrop` function is the output of a previous invocation. The Markov chain continues where the previous run stopped, doing just what it would have done if it had kept going, the initial state and random seed being the final state and final random seed of the previous invocation. Everything stays the same except for the arguments supplied (here `scale`).

- The argument `scale` controls the size of the Metropolis "normal random walk" proposal. The default is `scale = 1`. Big steps give lower acceptance rates. Small steps give higher. We want something about 20%. It is also possible to make `scale` a vector or a matrix. See `help(metrop)`.

Because each run starts where the last one stopped (when the first argument to `metrop` is the output of the previous invocation), each run serves as "burn-in" for its successor (assuming that any part of that run was worth anything at all).

3 Diagnostics

O. K. That does it for the acceptance rate. So let's do a longer run and look at the results.

```
> out <- metrop(out, nbatch = 1e4)
> t.test(out$accept.batch)$conf.int

[1] 0.2261945 0.2428055
attr(,"conf.level")
[1] 0.95

> out$time

   user  system elapsed
  0.381   0.012   0.392
```

Here we do a Monte Carlo confidence interval for the true unknown acceptance rate (what we would see with an infinite Monte Carlo sample size).

Figure 1 (page 6) shows the time series plot made by the R statement

```
> plot(ts(out$batch))
```

Another way to look at the output is an autocorrelation plot. Figure 2 (page 7) shows the time series plot made by the R statement

```
> acf(out$batch)
```

As with any multiplot plot, these are a bit hard to read. Readers are invited to make the separate plots to get a better picture. As with all "diagnostic" plots in MCMC, these don't "diagnose" subtle problems.

> The purpose of regression diagnostics is to find obvious, gross, embarrassing problems that jump out of simple plots (Geyer, 2006).

The time series plots will show *obvious* nonstationarity. They will not show *nonobvious* nonstationarity. They provide no guarantee whatsoever that your Markov chain is sampling anything remotely resembling the correct stationary distribution (with log unnormalized density lupost). In this very easy problem, we do not expect any convergence difficulties and so believe what the diagnostics seem to show, but one is a fool to trust such diagnostics in difficult problems.

The autocorrelation plots seem to show that the the autocorrelations are negligible after about lag 25. This diagnostic inference is reliable if the sampler is actually working (has nearly reached equilibrium) and worthless otherwise. Thus batches of length 25 should be sufficient, but let's use length 100 to be safe.

A more judicious discussion of asymptotics is found in Geyer (2011, Section 1.11.5), which says
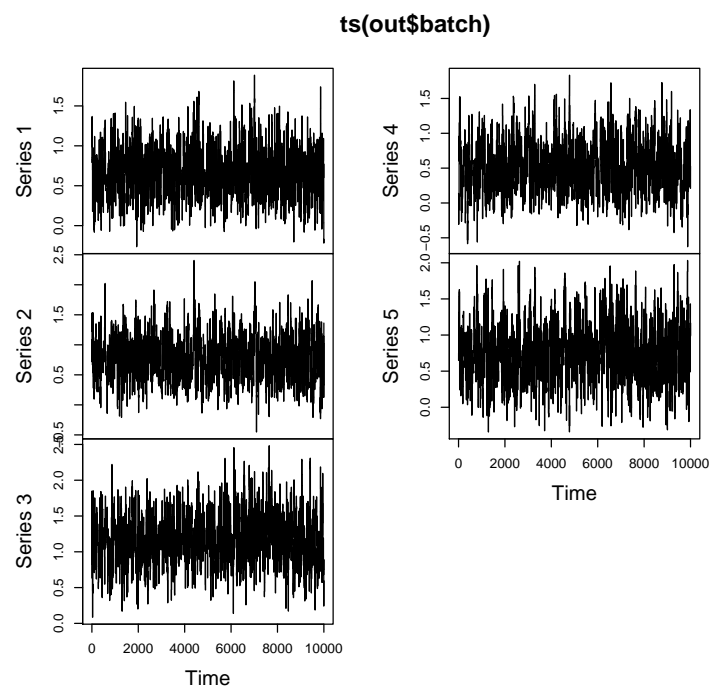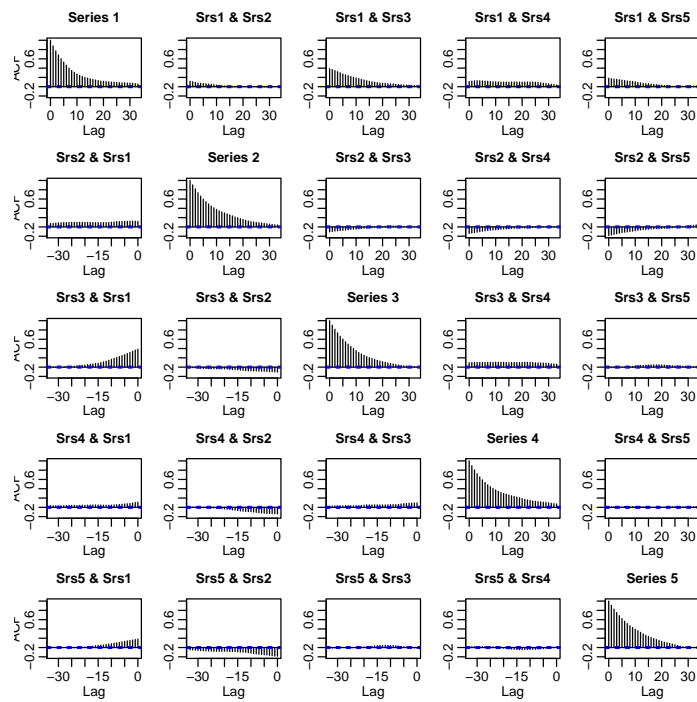
Figure 1: Time series plot of MCMC output.

Figure 2: Autocorrelation plot of MCMC output.

Many [diagnostics] come with theorems, but the theorems never prove the property you really want a diagnostic to have. These theorems say that if the chain converges, then the diagnostic will probably say that the chain converged, but they do not say that if the chain pseudo-converges, then the diagnostic will probably say that the chain did not converge.

4 Monte Carlo Estimates and Standard Errors

```
> out <- metrop(out, nbatch = 100, blen = 100,
+     outfun = function(z) c(z, z^2))
> t.test(out$accept.batch)$conf.int

[1] 0.222428 0.243972
attr(,"conf.level")
[1] 0.95

> out$time

   user  system elapsed
  0.412   0.000   0.411
```

We have added an argument `outfun` that gives the functional of the Markov chain (Geyer, 2011, Section 1.6) we want to average. For this problem we are interested in both posterior mean and variance. Mean is easy, just average the variables in question. But variance is a little tricky. We need to use the identity

$$\text{var}(X) = E(X^2) - E(X)^2$$

to write variance as a function of two things that can be estimated by simple averages. Hence we want to average the state itself and the squares of each component. Hence our `outfun` returns `c(z, z^2)` for an argument (the state vector) `z`.

4.1 Simple Means

The grand means (means of batch means) are

```
> apply(out$batch, 2, mean)

 [1] 0.6531950 0.7920342 1.1701075 0.5077331 0.7488265
 [6] 0.5145751 0.7560775 1.4973807 0.3913837 0.7244162
```

The first 5 numbers are the Monte Carlo estimates of the posterior means. The second 5 numbers are the Monte Carlo estimates of the posterior ordinary second moments. We get the posterior variances by

8

```
> foo <- apply(out$batch, 2, mean)
> mu <- foo[1:5]
> sigmasq <- foo[6:10] - mu^2
> mu

[1] 0.6531950 0.7920342 1.1701075 0.5077331 0.7488265

> sigmasq

[1] 0.08791134 0.12875924 0.12822924 0.13359081 0.16367507
```

Monte Carlo standard errors (MCSE) are calculated from the batch means. This is simplest for the means.

```
> mu.mcse <- apply(out$batch[ , 1:5], 2, sd) / sqrt(out$nbatch)
> mu.mcse

[1] 0.01224260 0.01417916 0.01793129 0.01468594 0.01582040
```

The extra factor sqrt(out$nbatch) arises because the batch means have variance σ^2/b where b is the batch length, which is out$blen, whereas the overall means mu have variance σ^2/n where n is the total number of iterations, which is out$blen * out$nbatch.

4.2   Functions of Means

To get the MCSE for the posterior variances we apply the delta method. Let u_i denote the sequence of batch means of the first kind for one parameter and $\bar{u}$ the grand mean (the estimate of the posterior mean of that parameter), let v_i denote the sequence of batch means of the second kind for the same parameter and $\bar{v}$ the grand mean (the estimate of the posterior second absolute moment of that parameter), and let $\mu = E(\bar{u})$ and $\nu = E(\bar{v})$. Then the delta method linearizes the nonlinear function

$$g(\mu, \nu) = \nu - \mu^2$$

as

$$\Delta g(\mu, \nu) = \Delta\nu - 2\mu\Delta\mu$$

saying that

$$g(\bar{u}, \bar{v}) - g(\mu, \nu)$$

has the same asymptotic normal distribution as

$$(\bar{v} - \nu) - 2\mu(\bar{u} - \mu)$$

which, of course, has variance 1 / nbatch times that of

$$(v_i - \nu) - 2\mu(u_i - \mu)$$

9

and this variance is estimated by

$$\frac{1}{n_{\text{batch}}} \sum_{i=1}^{n_{\text{batch}}} \left[(v_i - \bar{v}) - 2\bar{u}(u_i - \bar{u}) \right]^2$$

So

```
> u <- out$batch[ , 1:5]
> v <- out$batch[ , 6:10]
> ubar <- apply(u, 2, mean)
> vbar <- apply(v, 2, mean)
> deltau <- sweep(u, 2, ubar)
> deltav <- sweep(v, 2, vbar)
> foo <- sweep(deltau, 2, ubar, "*")
> sigmasq.mcse <- sqrt(apply((deltav - 2 * foo)^2, 2, mean) / out$nbatch)
> sigmasq.mcse

[1] 0.004241637 0.007292224 0.007271390 0.007374833
[5] 0.008175832
```

does the MCSE for the posterior variance.

Let's just check that this complicated `sweep` and `apply` stuff does do the right thing.

```
> sqrt(mean((((v[ , 2] - vbar[2]) - 2 * ubar[2] * (u[ , 2] - ubar[2]))^2) /
+     out$nbatch)

[1] 0.007292224
```

Comment Through version 0.5 of this vignette it contained an incorrect procedure for calculating this MCSE, justified by a handwave (which was incorrect). Essentially, it said to use the standard deviation of the batch means called `v` here, which appears to be very conservative.

4.3   Functions of Functions of Means

If we are also interested in the posterior standard deviation (a natural question, although not asked on the exam problem), the delta method gives its standard error in terms of that for the variance

```
> sigma <- sqrt(sigmasq)
> sigma.mcse <- sigmasq.mcse / (2 * sigma)
> sigma

[1] 0.2964985 0.3588304 0.3580911 0.3655008 0.4045678

> sigma.mcse

[1] 0.007152882 0.010161102 0.010152989 0.010088669
[5] 0.010104403
```

5 A Final Run

So that's it. The only thing left to do is a little more precision (the exam problem directed "use a long enough run of your Markov chain sampler so that the MCSE are less than 0.01")

```
> out <- metrop(out, nbatch = 500, blen = 400)
> t.test(out$accept.batch)$conf.int

[1] 0.232917 0.237393
attr(,"conf.level")
[1] 0.95

> out$time

   user   system elapsed
  8.046    0.000    8.046

> foo <- apply(out$batch, 2, mean)
> mu <- foo[1:5]
> sigmasq <- foo[6:10] - mu^2
> mu

[1] 0.6624650 0.7941013 1.1712710 0.5066326 0.7261414

> sigmasq

[1] 0.09189246 0.13323054 0.13230811 0.12871293 0.15978638

> mu.mcse <- apply(out$batch[ , 1:5], 2, sd) / sqrt(out$nbatch)
> mu.mcse

[1] 0.002960128 0.003647420 0.003787855 0.003632080
[5] 0.004273624

> u <- out$batch[ , 1:5]
> v <- out$batch[ , 6:10]
> ubar <- apply(u, 2, mean)
> vbar <- apply(v, 2, mean)
> deltau <- sweep(u, 2, ubar)
> deltav <- sweep(v, 2, vbar)
> foo <- sweep(deltau, 2, ubar, "*")
> sigmasq.mcse <- sqrt(apply((deltav - 2 * foo)^2, 2, mean) / out$nbatch)
> sigmasq.mcse

[1] 0.001044801 0.001725900 0.001518322 0.001553468
[5] 0.002010717
```

```
> sigma <- sqrt(sigmasq)
> sigma.mcse <- sigmasq.mcse / (2 * sigma)
> sigma

[1] 0.3031377 0.3650076 0.3637418 0.3587658 0.3997329

> sigma.mcse

[1] 0.001723311 0.002364198 0.002087087 0.002165017
[5] 0.002515076
```

and some nicer output, which is presented in three tables constructed from the R variables defined above using the R `xtable` command in the `xtable` library.

Table 1: Posterior Means

	constant	x_1	x_2	x_3	x_4
estimate	0.6625	0.7941	1.1713	0.5066	0.7261
MCSE	0.0030	0.0036	0.0038	0.0036	0.0043

Table 2: Posterior Variances

	constant	x_1	x_2	x_3	x_4
estimate	0.0919	0.1332	0.1323	0.1287	0.1598
MCSE	0.0010	0.0017	0.0015	0.0016	0.0020

Table 3: Posterior Standard Deviations

	constant	x_1	x_2	x_3	x_4
estimate	0.3031	0.3650	0.3637	0.3588	0.3997
MCSE	0.0017	0.0024	0.0021	0.0022	0.0025

Note for the record that the all the results presented in the tables are from "one long run" where long here took only 8.046 seconds (on whatever computer it was run on).

6   New Variance Estimation Functions

R function `initseq` (added in version 0.6 of this package) estimates variances in the Markov chain central limit theorem (CLT) following the methodology

introduced by Geyer (1992, Section 3.3). These methods only apply to scalar-valued functionals of reversible Markov chains, but the Markov chains produced by the `metrop` function satisfy this condition, even, as we shall see below, when batching is used.

Rather than redo the Markov chains in the preceding material, we just look at a toy problem, an AR(1) time series, which can be simulated in one line of R. This is the example on the help page for `initseq`.

```
> n <- 2e4
> rho <- 0.99
> x <- arima.sim(model = list(ar = rho), n = n)
```

The time series `x` is a reversible Markov chain and trivially a scalar-valued functional of a Markov chain.

Define

$$\gamma_k = \text{cov}(X_i, X_{i+k}) \tag{1}$$

where the covariances refer to the stationary Markov chain having the same transition probabilities as `x`. Then the variance in the CLT is

$$\sigma^2 = \gamma_0 + 2\sum_{k=1}^{\infty} \gamma_k$$

(Geyer, 1992, Theorem 2.1), that is,

$$\bar{x}_n \approx \text{Normal}\left(\mu, \frac{\sigma^2}{n}\right),$$

where $\mu = E(X_i)$ is the quantity being estimated by MCMC (in this toy problem we know $\mu = 0$).

Naive estimates of σ^2 obtained by plugging in empirical estimates of the gammas do not provide consistent estimation (Geyer, 1992, Section 3.1). Thus the scheme implemented by the R function `initseq`. Define

$$\Gamma_k = \gamma_{2k} + \gamma_{2k+1} \tag{2}$$

Geyer (1992, Theorem 3.1) says that Γ_k considered as a function of k is strictly positive, strictly decreasing, and strictly convex (provided we are, as stated above, working with a reversible Markov chain). Thus it makes sense to use estimators that use these properties. The estimators implemented by the R function `initseq` and described by Geyer (1992, Section 3.3) are conservative-consistent in the sense of Theorem 3.2 of that section.

Figure 3 (page 14) shows the time series plot made by the R statement

```
> out <- initseq(x)
> plot(seq(along = out$Gamma.pos) - 1, out$Gamma.pos,
+          xlab = "k", ylab = expression(Gamma[k]), type = "l")
> lines(seq(along = out$Gamma.dec) - 1, out$Gamma.dec, lty = "dotted")
> lines(seq(along = out$Gamma.con) - 1, out$Gamma.con, lty = "dashed")
```
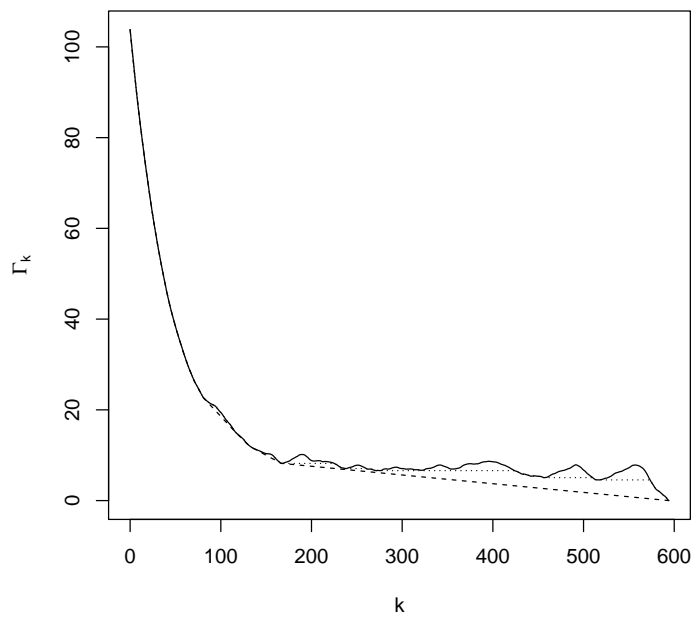
Figure 3: Plot "Big Gamma" defined by (1) and (2). Solid line, initial positive sequence estimator. Dotted line, initial monotone sequence estimator. Dashed line, initial convex sequence estimator.

One can use whichever curve one chooses, but now that the `initseq` function makes the computation trivial, it makes sense to use the initial convex sequence.

Of course, one is not interested in Figure 3, except perhaps when explaining the methodology. What is actually important is the estimate of σ^2, which is given by

```
> out$var.con

[1] 14386.32

> (1 + rho) / (1 - rho) * 1 / (1 - rho^2)

[1] 10000
```

where for comparison we have given the exact theoretical value of σ^2, which, of course, is never available in a non-toy problem.

These initial sequence estimators seem, at first sight to be a competitor for the method of batch means. However, appearances can be deceiving. The two methods are complementary. The sequence of batch means is itself a scalar-valued functional of a reversible Markov chain. Hence the initial sequence estimators can be applied to it.

```
> blen <- 5
> x.batch <- apply(matrix(x, nrow = blen), 2, mean)
> bout <- initseq(x.batch)
```

Because the batch length is too short, the variance of the batch means does not estimate σ^2. We must account for the autocorrelation of the batches, shown in Figure 4.

```
> plot(seq(along = bout$Gamma.con) - 1, bout$Gamma.con,
+          xlab = "k", ylab = expression(Gamma[k]), type = "l")
```

Because the the variance is proportional to one over the batch length, we need to multiply by the batch length to estimate the σ^2 for the original series.

```
> out$var.con

[1] 14386.32

> bout$var.con * blen

[1] 14466.75
```

Another way to look at this is that the MCMC estimator of μ is either `mean(x)` or `mean(x.batch)`. And the variance must be divided by the sample size to give standard errors. So either

```
> mean(x) + c(-1, 1) * qnorm(0.975) * sqrt(out$var.con / length(x))
```
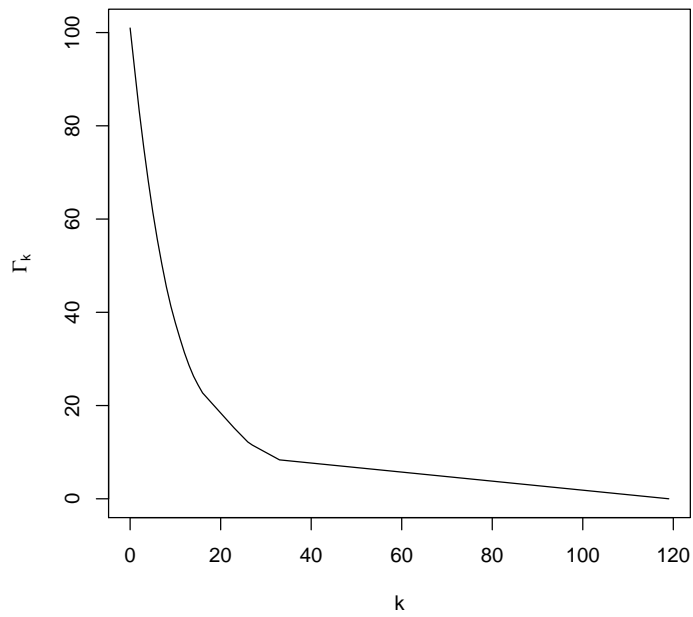
15

Figure 4: Plot "Big Gamma" defined by (1) and (2) for the sequence of batch means (batch length 5). Only initial convex sequence estimator is shown.

```
[1] -0.5078635   2.8167249
> mean(x.batch) + c(-1, 1) * qnorm(0.975) * sqrt(bout$var.con / length(x.batch))
[1] -0.5125039   2.8213653
```

is an asymptotic 95% confidence interval for μ. Just divide by the relevant sample size.

A Dot-dot-dot Versus Global Variables Versus Closures

This appendix deals with three ways to pass information to a function being passed to another R function (a higher-order function), for example when

- the function being passed is the objective function for an optimization done by the higher-order function, which optimizes (R function `optim` for example),

- the function being passed is the integrand for an integration done by the higher-order function, which integrates (R function `integrate` for example),

- the function being passed is the estimator of a parameter for a bootstrap done by the higher-order function, which simulates the (bootstrap approximation) of the sampling distribution of the estimator (R function `boot` in R package `boot`, for example),

- the function being passed is the log unnormalized density function for a simulation done by the higher-order function, which simulates the distribution having that unnormalized density (R function `metrop` in this package for example),

These ways are

- using dot-dot-dot (R syntax ...),

- using global variables,

- using closures, also called the function factory pattern.

The main body of this vignette uses the third option. This appendix explains them all and the virtues and vices of each.

A.1 Dot-dot-dot

The dot-dot-dot mechanism is fairly easy to use when only one function is passed to the higher-order function, but does require care and more work to use. It is even harder to deal with when more than one function is passed to the higher-order function. R functions `metrop` and `temper` in this package can be passed two functions: the log unnormalized density function and the output function.

A.2 Only One Function Argument

Previous versions of this vignette used the dot-dot-dot mechanism everywhere. In those versions the log unnormalized density function was defined by

```
> lupost <- function(beta, x, y) {
+       eta <- as.numeric(x %*% beta)
+       logp <- ifelse(eta < 0, eta - log1p(exp(eta)), - log1p(exp(- eta)))
+       logq <- ifelse(eta < 0, - log1p(exp(eta)), - eta - log1p(exp(- eta)))
+       logl <- sum(logp[y == 1]) + sum(logq[y == 0])
+       return(logl - sum(beta^2) / 8)
+ }
```

rather than the way it is now done in the main text above. Note that everything is the same in the two definitions except here we have

```
lupost <- function(beta, x, y) {
```

where in the main text we now have

```
lupost_factory <- function(x, y) function(beta) {
```

and then we have to execute the function factory `lupost_factory` to make the `lupost` function.

But the main difference is that R function `lupost` (the user written function specifying the log unnormalized posterior density)

- here has 3 arguments, `beta`, `x`, and `y`, and the latter two must be passed via the dot-dot-dot mechanism, but

- there has 1 argument, `beta`, and it just knows about `x` and `y` — they are in its closure.

So to use this `lupost` function, we have to add arguments `x` and `y` to each call to R function `metrop`. For example

```
> out <- glm(y ~ x1 + x2 + x3 + x4, data = logit,
+       family = binomial, x = TRUE)
> x <- out$x
> y <- out$y
> out <- metrop(lupost, beta.init, 1e3, x = x, y = y)
> out$accept

[1] 0.015

> out <- metrop(out, scale = 0.1, x = x, y = y)
> out$accept

[1] 0.759
```

18

```
> out <- metrop(out, scale = 0.3, x = x, y = y)
> out$accept

[1] 0.368

> out <- metrop(out, scale = 0.5, x = x, y = y)
> out$accept

[1] 0.174

> out <- metrop(out, scale = 0.4, x = x, y = y)
> out$accept

[1] 0.245
```

and so forth.

This method has the benefit that we do not have to explain function factories and has the drawback that we need to keep remembering to add `x = x, y = y` to each invocation of R function `metrop`. It is unclear to me which pattern is more mysterious to naive users.

A.3   More Than One Function Argument

The situation becomes more complicated (see the Warning section of the help pages for R functions `metrop` and `temper`) when more than one function argument is passed to the higher-order function. Then *they all must handle the same dot-dot-dot arguments* whether or not they want them.

So now we must define the output function as

```
> outfun <- function(z, ...) c(z, z^2)
```

The `...` argument in the function signature is essential because this function is going to be passed dot-dot-dot arguments `x` and `y`, which it does not need and does not want, so it has to allow for them (and then not use them).

Then we can continue

```
> out <- metrop(out, nbatch = 100, blen = 100, outfun = outfun,
+     x = x, y = y)
> out$accept

[1] 0.2386
```

and this works. But we would have gotten an error about unused arguments if we had defined the output function without the dot-dot-dot as we did in the main text.

Your humble author was himself confused about this for years and so doubts that naive R users will find it intuitive.

A.4 Global Variables

As every programmer knows global variables are easy to use and evil (C2 Wiki, 2013). They are part of the way of R (or perhaps part of one of the ways of R). They are OK for "very small or one-off programs" (C2 Wiki, 2013). If you are going to throw the code away after one use, fine. If you are going to give your code to other people or even use it yourself six months later (after you have long forgotten the details), then this method is evil and should be avoided.

In this method we define both functions passed to the higher-order function without ... and without using a function factory. We already in the preceding section of this appendix defined R objects x and y as global variables (in the R global environment .GlobalEnv). They are global variables and we use them as such, defining

```
> lupost <- function(beta) {
+       eta <- as.numeric(x %*% beta)
+       logp <- ifelse(eta < 0, eta - log1p(exp(eta)), - log1p(exp(- eta)))
+       logq <- ifelse(eta < 0, - log1p(exp(eta)), - eta - log1p(exp(- eta)))
+       logl <- sum(logp[y == 1]) + sum(logq[y == 0])
+       return(logl - sum(beta^2) / 8)
+ }
> outfun <- function(z) c(z, z^2)
```

Then the following works

```
> out <- metrop(lupost, beta.init, 1e3)
> out$accept

[1] 0.032

> out <- metrop(out, scale = 0.1)
> out$accept

[1] 0.764

> out <- metrop(out, scale = 0.3)
> out$accept

[1] 0.346

> out <- metrop(out, scale = 0.5)
> out$accept

[1] 0.142

> out <- metrop(out, scale = 0.4)
> out$accept

[1] 0.247
```

```
> out <- metrop(out, nbatch = 100, blen = 100, outfun = outfun)
> out$accept
```

[1] 0.2298

We get the best of both worlds. We don't need `x = x, y = y` and we don't need a function factory.

But if we change the name of the global variables to say `modmat` and `resp` instead of `x` and `y`, then our code breaks. R function `lupost` is looking up global variables `x` and `y` *under those names* not under any other names. So your code using this method is rigid and brittle and probably unusable by others.

Note that if we are using either of the other methods, renaming is no problem. Using dot-dot-dot we do

```
out <- metrop(out, scale = 0.1, x = modmat, y = resp)
```

Using the function factory we do

```
lupost <- lupost_factory(modmat, resp)
```

See Section 7.5.3 of Geyer (2018) for more about this.

A.5   Function Factory

The terminology "function factory pattern" is apparently due to Wickham (2014, Section 10.3.1). But it is just a special case about how closures work (in R and all other languages that have them). Compare

```
> fred <- function(y) function(x) x + y
> fred(2)(3)
```

[1] 5

(Geyer, 2018, Section 6.5)) to

```
> lupost_factory <- function(x, y) function(beta) {
+     eta <- as.numeric(x %*% beta)
+     logp <- ifelse(eta < 0, eta - log1p(exp(eta)), - log1p(exp(- eta)))
+     logq <- ifelse(eta < 0, - log1p(exp(eta)), - eta - log1p(exp(- eta)))
+     logl <- sum(logp[y == 1]) + sum(logq[y == 0])
+     return(logl - sum(beta^2) / 8)
+ }
> lupost <- lupost_factory(x, y)
> lupost(beta.init)
```

[1] -44.19627

We could also do the same calculation treating `lupost_factory` as just an ordinary curried function, like R function `fred` in the preceding example,

```
> lupost_factory(x, y)(beta.init)
```

```
[1] -44.19627
```

So there is nothing mysterious about function factories. They are just one particular use of the essence of functional programming (closures). If you really understand functional programming, then you must understand this.

Long ago, when I switched from S to R, I would not have considered that being an "knowledgeable user" of R required knowledge of closures. Now I do. This is partly because other functional languages like Scheme, Javascript, F#, Clojure, and Haskell have become very popular for general computer programming. So now we want to emphasize that we can do in R what we can do in them. But it is also partly because I now understand more about functional programming. I can now see that if you really understand closures, then you don't need most other features of these programming languages (including R). Following Crockford (2008) we can say that all programming languages (including R) have their good parts and their bad parts and we should use the good features and avoid the bad features. The best feature is closures. Crockford calls this the best idea ever put in a programming language. So we should use it a lot.

If I were starting to write the mcmc package today, I might leave out the dot-dot-dot arguments of R functions metrop, morph.metrop, and temper. That would mean that users could not use the dot-dot-dot pattern. They would be forced to use the function factory pattern or the global variables pattern. And if they had accepted that the global variables pattern is evil, then they would have to use the function factory pattern. (If you like the dot-dot-dot pattern, don't worry. It is not going to be removed from this package. Backward compatibility trumps everything.)

References

C2 Wiki Contributors (2013). Global Variables Are Bad. `http://wiki.c2.com/?GlobalVariablesAreBad`.

Crockford, D. (2008). *JavaScript: The Good Parts*. O'Reilly, Sebastopol CA.

Gelman, A., G. O. Roberts, and W. R. Gilks (1996). Efficient Metropolis jumping rules. In *Bayesian Statistics, 5 (Alicante, 1994)*, pp. 599–607. Oxford University Press.

Geyer, C. J. (1992). Practical Markov chain Monte Carlo (with discussion). *Statistical Science*, 7, 473–511.

Geyer, C. J. (2006). On the bogosity of MCMC diagnostics. `http://users.stat.umn.edu/~geyer/mcmc/diag.html`.

Geyer, C. J. (2011). Introduction to Markov chain Monte Carlo. In *Handbook of Markov Chain Monte Carlo*, edited by Brooks, S., Gelman, A., Jones, G., and Meng, X.-L., pp. 3–48. Chapman & Hall/CRC, Boca Raton, FL.

Geyer, C. J. (2018). Stat 3701 lecture notes: Basics of R. `http://www.stat.umn.edu/geyer/3701/notes/basic.html`.

Geyer, C. J. and E. A. Thompson (1995). Annealing Markov chain Monte Carlo with applications to ancestral inference. *Journal of the American Statistical Association*, 90, 909–920.

Wickham, H. (2014). *Advanced R.* Chapman & Hall/CRC, Boca Raton, FL. Also available on the web `http://adv-r.had.co.nz/`.