Bayes Factors via Serial Tempering

Charles J. Geyer

April 15, 2017

1 Introduction

1.1 Bayes Factors

Let $\mathcal{M}$ be a finite or countable set of models (here we only deal with finite $\mathcal{M}$ but Bayes factors make sense for countable $\mathcal{M}$). For each model $m \in \mathcal{M}$ we have the prior probability of the model $\text{pri}(m)$. It does not matter if this prior on models is unnormalized.

Each model m has a parameter space Θ_m and a prior

$$g(\theta \mid m), \qquad \theta \in \Theta_m$$

The spaces Θ_m can and usually do have different dimensions. That's the point. These within model priors must be normalized proper priors. The calculations to follow make no sense if these priors are unnormalized or improper.

Each model m has a data distribution

$$f(y \mid \theta, m)$$

and the observed data y may be either discrete or continuous (it makes no difference to the Bayesian who treats y as fixed after it is observed and treats only θ and m as random).

The unnormalized posterior for everything (for models and parameters within models) is

$$f(y \mid \theta, m) g(\theta \mid m) \, \text{pri}(m)$$

To obtain the conditional distribution of y given m, we must integrate out the nuisance parameter θ

$$\begin{aligned} q(y \mid m) &= \int_{\Theta_m} f(y \mid \theta, m) g(\theta \mid m) \, \text{pri}(m) \, d\theta \\ &= \text{pri}(m) \int_{\Theta_m} f(y \mid \theta, m) g(\theta \mid m) \, d\theta \end{aligned}$$

These are the unnormalized posterior probabilities of the models. The normalized posterior probabilities are

$$\operatorname{post}(m \mid y) = \frac{q(y \mid m)}{\sum_{m \in \mathcal{M}} q(y \mid m)}$$

It is considered useful to define

$$b(y \mid m) = \int_{\Theta_m} f(y \mid \theta, m) g(\theta \mid m) \, d\theta$$

so

$$q(y \mid m) = b(y \mid m) \operatorname{pri}(m)$$

Then the ratio of posterior probabilities of models m_1 and m_2 is

$$\frac{\operatorname{post}(m_1 \mid y)}{\operatorname{post}(m_2 \mid y)} = \frac{q(y \mid m_1)}{q(y \mid m_2)} = \frac{b(y \mid m_1)}{b(y \mid m_2)} \cdot \frac{\operatorname{pri}(m_1)}{\operatorname{pri}(m_2)}$$

This ratio is called the *posterior odds* of the models (a ratio of probabilities is called an *odds*) of these models.

The *prior odds* is

$$\frac{\operatorname{pri}(m_1)}{\operatorname{pri}(m_2)}$$

The term we have not yet named in

$$\frac{\operatorname{post}(m_1 \mid y)}{\operatorname{post}(m_2 \mid y)} = \frac{b(y \mid m_1)}{b(y \mid m_2)} \cdot \frac{\operatorname{pri}(m_1)}{\operatorname{pri}(m_2)}$$

is called the *Bayes factor*

$$\frac{b(y \mid m_1)}{b(y \mid m_2)} \tag{1}$$

the ratio of posterior odds to prior odds.

The prior odds tells how the prior compares the probability of the models. The Bayes factor tells us how the data shifts that comparison going from prior to posterior via Bayes rule. Bayes factors are the primary tool Bayesians use for model comparison, the competitor for frequentist P-values in frequentist hypothesis tests of model comparison.

Note that our clumsy multiple letter notation for priors and posteriors $\operatorname{pri}(m)$ and $\operatorname{post}(m \mid y)$ does not matter because neither is involved in the actual calculation of Bayes factors (1). Priors and posteriors are involved in motivating Bayes factors but not in calculating them.

1.2 Tempering

Simulated tempering (Marinari and Parisi, 1992; Geyer and Thompson, 1995) is a method of Markov chain Monte Carlo (MCMC) simulation of many distributions at once. It was originally invented with the primary aim of speeding up MCMC convergence, but was also recognized to be useful for sampling multiple distributions (Geyer and Thompson, 1995). In the latter role it is sometimes referred to as "umbrella sampling" which is a term coined by Torrie and Valleau (1977) for sampling multiple distributions via MCMC.

We have a finite set of unnormalized distributions we want to sample, all related in some way. The R function `temper` in the CRAN package `mcmc` requires all to have continuous distributions for random vectors of the same dimension (all distributions have the same domain $\mathbb{R}^p$). Let h_i, $i \in \mathcal{I}$ denote the unnormalized densities of these distributions. Simulated tempering (called "serial tempering" by the `temper` function to distinguish from a related scheme not used in this document called "parallel tempering" and in either case abbreviated ST) runs a Markov chain whose state is a pair (i, x) where $i \in \mathcal{I}$ and $x \in \mathbb{R}^p$.

The unnormalized density of stationary distribution of the ST chain is

$$h(i, x) = h_i(x) c_i \tag{2}$$

where the c_i are arbitrary constants chosen by the user (more on this later).

The equilibrium distribution of the ST state (I, X) — both bits random — is such that conditional distribution of X given $I = i$ is the distribution with unnormalized density h_i. This is obvious from $h(i, x)$ being the unnormalized conditional density — the same function thought of as a function of both variables is the unnormalized joint density and thought of as a function of just one of the variables is an unnormalized conditional density — and $h(i, x)$ thought of as a function of x for fixed i being proportional to h_i. The equilibrium unnormalized marginal distribution of I is

$$\int h(i, x)\, dx = c_i \int h_i(x)\, dx = c_i d_i \tag{3}$$

where

$$d_i = \int h_i(x)\, dx$$

is the normalizing constant for h_i, that is, h_i / d_i is a normalized distribution.

It is clear from (3) being the unnormalized marginal distribution that in order for the marginal distribution to be uniform we must choose the tuning

constants c_i to be proportional to $1/d_i$. It is not important that the marginal distribution be exactly uniform, but unless it is approximately uniform, the sampler will not visit each distribution frequently. Thus we do need to have the c_i to be approximately proportional to $1/d_i$. This is accomplished by trial and error (one example is done in this document) and is easy for easy problems and hard for hard problems (Geyer and Thompson, 1995, have much to say about adjusting the c_i). For the rest of this section we will assume the tuning constants c_i have been so adjusted: we do not have the c_i exactly proportional to $1/d_i$ but do have them approximately proportional to $1/d_i$.

1.3 Tempering and Bayes Factors

Bayes factors are very important in Bayesian inference and many methods have been invented to calculate them. No method except the one described here using ST is anywhere near as accurate and straightforward. Thus no competitors will be discussed.

In using ST for Bayes factors we identify the index set $\mathcal{I}$ with the model set $\mathcal{M}$ and use the integers $1, \ldots, k$ for both. We would like to identify the within model parameter vector θ with the vector x that is the continuous part of the state of the ST Markov chain, but cannot because the dimension of θ depends on m and this is not allowed. Thus we have to do something a bit more complicated. We "pad" θ so that it always has the same dimension, doing so in a way that does not interfere with the Bayes factor calculation. Write $\theta = (\theta_{\text{actual}}, \theta_{\text{pad}})$, the dimension of both parts depending on the model m. Then we insist on the following conditions:

$$f(y \mid \theta, m) = f(y \mid \theta_{\text{actual}}, m)$$

so the data distribution does not depend on the "padding" and

$$g(\theta \mid m) = g_{\text{actual}}(\theta_{\text{actual}} \mid m) \cdot g_{\text{pad}}(\theta_{\text{pad}} \mid m)$$

so the two parts are *a priori* independent and both parts of the prior are normalized proper priors. This assures that

$$\begin{aligned} b(y \mid m) &= \int_{\Theta_m} f(y \mid \theta, m) g(\theta \mid m) \, d\theta \\ &= \iint f(y \mid \theta_{\text{actual}}, m) g_{\text{actual}}(\theta_{\text{actual}} \mid m) g_{\text{pad}}(\theta_{\text{pad}} \mid m) \, d\theta_{\text{actual}} \, d\theta_{\text{pad}} \\ &= \int_{\Theta_m} f(y \mid \theta_{\text{actual}}, m) g_{\text{actual}}(\theta_{\text{actual}} \mid m) \, d\theta_{\text{actual}} \end{aligned} \tag{4}$$

so the calculation of the unnormalized Bayes factors is the same whether or not we "pad" θ, and we may then take

$$\begin{aligned} h_m(\theta) &= f(y \mid \theta, m) g(\theta \mid m) \\ &= f(y \mid \theta_{\text{actual}}, m) g_{\text{actual}}(\theta_{\text{actual}} \mid m) g_{\text{pad}}(\theta_{\text{pad}} \mid m) \end{aligned}$$

to be the unnormalized densities for the component distributions of the ST chain, in which case the unnormalized Bayes factors are proportional to the normalizing constants d_i in Section 1.2.

1.4 Tempering and Normalizing Constants

Let d be the normalizing constant for the joint equilibrium distribution of the ST chain (2). When we are running the ST chain we know neither d nor the d_i but we do know the c_i, which are constants we have chosen based on the results of previous runs but are fixed known numbers for the current run. Let (I_t, X_t), $t = 1, 2, \ldots$ be the sample path of the ST chain. Recall that (somewhat annoyingly) we are using the notation (i, x) for the state vector of a general ST chain and the notation (m, θ) for ST chains used to calculate Bayes factors, identifying $i = m$ and $x = \theta$.

Let $\text{ind}(\cdot)$ denote the function that maps logical values to numerical values, false to zero and true to one. Normalizing constants are estimated by averaging the time spent in each model

$$\hat{\delta}_n(m) = \frac{1}{n} \sum_{t=1}^{n} \text{ind}(I_t = m) \tag{5}$$

For the purposes of approximating Bayes factors the X_t are ignored. The X_t may be useful for other purposes, such as Bayesian model averaging (Hoeting, Madigan, Raftery, and Volinsky, 1999), but this is not discussed here.

The Monte Carlo approximations (5) converge to their expected values under the equilibrium distribution

$$E\{\text{ind}(I_t = m)\} = \int \frac{h(m, x)}{d} \, dx = \frac{c_m d_m}{d} = \delta(m) \tag{6}$$

We want to estimate the unnormalized Bayes factors (4), which are in this context proportional to the d_m. The c_m are known, d is unknown but does not matter since we only need to estimate the $d_m = b(m \mid y)$ up to an overall unknown constant of proportionality, which cancels out of Bayes factors (1).

Note that our discussion here applies unchanged to the general problem of estimating normalizing constants up to an unknown constant of proportionality, which has applications other than Bayes factors, for example, missing data maximum likelihood (Thompson and Guo, 1991; Geyer, 1994; Sung and Geyer, 2007). The ST method approximates normalizing constants up to an overall constant of proportionality with high accuracy regardless of how large or small they are (whether they are 10^{100} or 10^{-100}), and no other method that does not use essentially the same idea can do this.

The key is what seems at first sight to be a weakness of ST, the need to adjust the tuning constants c_i by trial and error. In this context the weakness is actually a strength: the adjusted c_i contain most of the information about the size of the normalizing constants d_i and the Monte Carlo averages (5) add only the finishing touch. Thus multiple runs of the ST chain with different choices of the c_i used in each run are needed (the "trial and error"), but the information from all are incorporated in the final run used for final approximation of the normalizing constants (Bayes factors). It is perhaps surprising that the Monte Carlo error approximation is trivial. In the context of the last run of the ST chain the c_i are known constants and contribute no error. The Monte Carlo error of the averages (5) is straightforwardly estimated by batch means or competing methods.

Geyer and Thompson (1995) note that the c_i enter formally like a prior: one can think of $h_i(x)c_i$ as likelihood times prior. But one should not think of the c_i as representing prior information, informative, non-informative, or in between. The c_i are adjusted to make the ST distribution sample all the models h_i, and that is the only criterion for the adjustment. For this reason Geyer and Thompson (1995) call the c_i the *pseudoprior*. This is a special case of a general principle of MCMC. When doing MCMC one should forget the statistical motivation (in this case Bayes factors). One should set up a Markov chain that does a good job of simulating the required equilibrium distribution, whatever it is. Thinking about the statistical motivation of the equilibrium does not help and can hurt (if one thinks of the pseudoprior as an actual prior, one may be tempted to adjust it to represent prior information).

2 R Package MCMC

We use the R statistical computing environment (R Development Core Team, 2010) in our analysis. It is free software and can be obtained from `http://cran.r-project.org`. Precompiled binaries are available for Windows, Macintosh, and popular Linux distributions. We use the contributed

package `mcmc` (Geyer., 2009) If R has been installed, but this package has not yet been installed, do

```
install.packages("mcmc")
```

from the R command line (or do the equivalent using the GUI menus if on Apple Macintosh or Microsoft Windows). This may require root or administrator privileges.

Assuming the `mcmc` package has been installed, we load it

```
> library(mcmc)
```

The version of the package used to make this document is 0.9-5 (which is available on CRAN). The version of R used to make this document is 3.3.3.

We also set the random number generator seed so that the results are reproducible.

```
> set.seed(42)
```

To get different results, change the setting or don't set the seed at all.

3 Logistic Regression Example

We use the same logistic regression example used in the `mcmc` package vignette for the `metrop` function (file `demo.pdf`. Simulated data for the problem are in the data set `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

```
> 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 example isn't about 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.

Moreover, we wish to calculate Bayes factors for the $16 = 2^4$ possible submodels that include or exclude each of the predictors, `x1`, `x2`, `x3`, and `x4`.

3.1 Setup

We set up a matrix that indicates these models.

```
> varnam <- names(coefficients(out))
> varnam <- varnam[varnam != "(Intercept)"]
> nvar <- length(varnam)
> models <- NULL
> foo <- seq(0, 2^nvar - 1)
> for (i in 1:nvar) {
+     bar <- foo %/% 2^(i - 1)
+     bar <- bar %% 2
+     models <- cbind(bar, models, deparse.level = 0)
```

```
+ }
> colnames(models) <- varnam
> models

      x1 x2 x3 x4
 [1,]  0  0  0  0
 [2,]  0  0  0  1
 [3,]  0  0  1  0
 [4,]  0  0  1  1
 [5,]  0  1  0  0
 [6,]  0  1  0  1
 [7,]  0  1  1  0
 [8,]  0  1  1  1
 [9,]  1  0  0  0
[10,]  1  0  0  1
[11,]  1  0  1  0
[12,]  1  0  1  1
[13,]  1  1  0  0
[14,]  1  1  0  1
[15,]  1  1  1  0
[16,]  1  1  1  1
```

In each row, 1 indicates the predictor is in the model and 0 indicates it is out.

The function `temper` in the mcmc package that does tempering requires a notion of neighbors among models. It attempts jumps only between neighboring models. Here we choose models to be neighbors if they differ only by one predictor.

```
> neighbors <- matrix(FALSE, nrow(models), nrow(models))
> for (i in 1:nrow(neighbors)) {
+     for (j in 1:ncol(neighbors)) {
+         foo <- models[i, ]
+         bar <- models[j, ]
+         if (sum(foo != bar) == 1) neighbors[i, j] <- TRUE
+     }
+ }
```

Now we specify the equilibrium distribution of the ST chain. Its state vector is (i, x) or (m, θ) in our alternative notations, where i is an integer between 1 and `nrow(models)` $= 16$ and θ is the parameter vector "padded"

to always be the same length, so we take it to be the length of the parameter vector of the full model which is `length(out$coefficients)` or `ncol(models) + 1` which makes the length of the state of the ST chain `ncol(models) + 2`. We take the within model priors for the "padded" components of the parameter vector to be the same as those for the "actual" components, normal with mean 0 and standard deviation 2 for all cases. As is seen in (4) the priors for the "padded" components (parameters not in the model for the current state) do not matter because they drop out of the Bayes factor calculation. The choice does not matter much for this toy example. See the discussion section for more on this issue. It is important that we use normalized log priors, the term `dnorm(beta, 0, 2, log = TRUE)` in the function, unlike when we are simulating only one model as in the `mcmc` package vignette where it would be o. k. to use unnormalized log priors `- beta^2 / 8`. The `temper` function wants the log unnormalized density of the equilibrium distribution. We include an additional argument `log.pseudo.prior`, which is $\log(c_i)$ in our mathematical development, because this changes from run to run as we adjust it by trial and error. Other "arguments" are the model matrix of the full model `modmat`, the matrix `models` relating integer indices (the first component of the state vector of the ST chain) to which predictors are in or out of the model, and the data vector `y`, but these are not passed as arguments to our function and instead are found in the R global environment.

```
> modmat <- out$x
> y <- logit$y
> ludfun <- function(state, log.pseudo.prior) {
+     stopifnot(is.numeric(state))
+     stopifnot(length(state) == ncol(models) + 2)
+     icomp <- state[1]
+     stopifnot(icomp == as.integer(icomp))
+     stopifnot(1 <= icomp && icomp <= nrow(models))
+     stopifnot(is.numeric(log.pseudo.prior))
+     stopifnot(length(log.pseudo.prior) == nrow(models))
+     beta <- state[-1]
+     inies <- c(TRUE, as.logical(models[icomp, ]))
+     beta.logl <- beta
+     beta.logl[! inies] <- 0
+     eta <- as.numeric(modmat %*% beta.logl)
+     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])
+     logl + sum(dnorm(beta, 0, 2, log = TRUE)) + log.pseudo.prior[icomp]
+ }
```

3.2 Trial and Error

Now we are ready to try it out. We start in the full model at its MLE, and we initialize `log.pseudo.prior` at all zeros, having no idea *a priori* what it should be.

```
> state.initial <- c(nrow(models), out$coefficients)
> qux <- rep(0, nrow(models))
> out <- temper(ludfun, initial = state.initial, neighbors = neighbors,
+     nbatch = 1000, blen = 100, log.pseudo.prior = qux)
> names(out)

 [1] "lud"          "neighbors"    "nbatch"       "blen"
 [5] "nspac"        "scale"        "outfun"       "debug"
 [9] "parallel"     "initial.seed" "final.seed"   "time"
[13] "batch"        "acceptx"      "accepti"      "initial"
[17] "final"        "ibatch"

> out$time

   user  system elapsed
 10.428   0.000  10.439
```

So what happened?

```
> ibar <- colMeans(out$ibatch)
> ibar

 [1] 0.00000 0.00000 0.00000 0.00000 0.00524 0.06489 0.00754
 [8] 0.06021 0.00033 0.00202 0.00008 0.00054 0.28473 0.31487
[15] 0.12478 0.13477
```

The ST chain did not mix well, several models not being visited even once. So we adjust the pseudo priors to get uniform distribution.

```
> qux <- qux + pmin(log(max(ibar) / ibar), 10)
> qux <- qux - min(qux)
> qux
```

```
 [1] 10.0000000 10.0000000 10.0000000 10.0000000  4.0958384
 [6]  1.5794663  3.7319377  1.6543214  6.8608225  5.0490623
[11]  8.2778885  6.3683460  0.1006185  0.0000000  0.9256077
[16]  0.8485902
```

The new pseudoprior should be proportional to `1 / ibar` if `ibar` is an accurate estimate of (6), but this makes no sense when the estimates are bad, in particular, when the are exactly zero. Thus we put an upper bound, chosen arbitrarily (here 10) on the maximum increase of the log pseudoprior. The statement

```
qux <- qux - min(qux)
```

is unnecessary. An overall arbitrary constant can be added to the log pseudoprior without changing the equilibrium distribution of the ST chain. We do this only to make `qux` more comparable from run to run.

Now we repeat this until the log pseudoprior "converges" roughly. Because this loop takes longer than CRAN vingettes are supposed to take, we save the results to a file and load the results from this file if it already exists.

```
> lout <- suppressWarnings(try(load("bfst1.rda"), silent = TRUE))
> if (inherits(lout, "try-error")) {
+     qux.save <- qux
+     time.save <- out$time
+     repeat{
+         out <- temper(out, log.pseudo.prior = qux)
+         ibar <- colMeans(out$ibatch)
+         qux <- qux + pmin(log(max(ibar) / ibar), 10)
+         qux <- qux - min(qux)
+         qux.save <- rbind(qux.save, qux, deparse.level = 0)
+         time.save <- rbind(time.save, out$time, deparse.level = 0)
+         if (max(ibar) / min(ibar) < 2) break
+     }
+     save(out, qux, qux.save, time.save, file = "bfst1.rda")
+ } else {
+     .Random.seed <- out$final.seed
+ }
> print(qux.save, digits = 3)

     [,1]  [,2] [,3]  [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11]
[1,] 10.0 10.00 10.0 10.00 4.10 1.58 3.73 1.65 6.86  5.05  8.28
```

```
[2,] 17.7 10.00 14.4  9.71 4.05 1.54 3.35 1.83 6.02  4.41  6.61
[3,] 18.8  9.33 14.4  9.01 3.97 1.39 3.16 1.40 5.64  4.15  6.33
[4,] 18.9  9.73 14.7  9.48 4.28 1.49 3.23 1.55 6.19  4.62  6.88
     [,12] [,13]   [,14] [,15] [,16]
[1,]  6.37 0.101 0.00000 0.926 0.849
[2,]  5.68 0.148 0.00000 0.737 0.787
[3,]  5.17 0.000 0.00324 0.441 0.614
[4,]  5.66 0.110 0.00000 0.915 0.833
```

```
> print(qux, digits = 3)

 [1] 18.947  9.733 14.714  9.478  4.276  1.491  3.230  1.553
 [9]  6.187  4.624  6.881  5.660  0.110  0.000  0.915  0.833
```

```
> apply(time.save, 2, sum)

 user.self   sys.self    elapsed user.child  sys.child
    90.701      0.000     91.010      0.000      0.000
```

Now that the pseudoprior is adjusted well enough, we need to perhaps make other adjustments to get acceptance rates near 20%.

```
> print(out$accepti, digits = 3)

       [,1]  [,2]  [,3]  [,4]  [,5]  [,6]  [,7]  [,8]  [,9]
 [1,]    NA 0.239 0.212    NA 0.221    NA    NA    NA 0.218
 [2,] 0.294    NA    NA 0.250    NA 0.193    NA    NA    NA
 [3,] 0.236    NA    NA 0.261    NA    NA 0.199    NA    NA
 [4,]    NA 0.278 0.329    NA    NA    NA    NA 0.216    NA
 [5,] 0.220    NA    NA    NA    NA 0.287 0.348    NA    NA
 [6,]    NA 0.155    NA    NA 0.235    NA    NA 0.294    NA
 [7,]    NA    NA 0.158    NA 0.290    NA    NA 0.250    NA
 [8,]    NA    NA    NA 0.147    NA 0.312 0.254    NA    NA
 [9,] 0.344    NA    NA    NA    NA    NA    NA    NA    NA
[10,]    NA 0.230    NA    NA    NA    NA    NA    NA 0.242
[11,]    NA    NA 0.233    NA    NA    NA    NA    NA 0.213
[12,]    NA    NA    NA 0.250    NA    NA    NA    NA    NA
[13,]    NA    NA    NA    NA 0.243    NA    NA    NA 0.171
[14,]    NA    NA    NA    NA    NA 0.235    NA    NA    NA
[15,]    NA    NA    NA    NA    NA    NA 0.364    NA    NA
[16,]    NA    NA    NA    NA    NA    NA    NA 0.272    NA
```

```
      [,10] [,11] [,12] [,13] [,14] [,15] [,16]
 [1,]    NA    NA    NA    NA    NA    NA    NA
 [2,] 0.230    NA    NA    NA    NA    NA    NA
 [3,]    NA 0.207    NA    NA    NA    NA    NA
 [4,]    NA    NA 0.250    NA    NA    NA    NA
 [5,]    NA    NA    NA 0.319    NA    NA    NA
 [6,]    NA    NA    NA    NA 0.237    NA    NA
 [7,]    NA    NA    NA    NA    NA 0.229    NA
 [8,]    NA    NA    NA    NA    NA    NA 0.251
 [9,] 0.251 0.185    NA 0.254    NA    NA    NA
[10,]    NA    NA 0.277    NA 0.220    NA    NA
[11,]    NA    NA 0.286    NA    NA 0.217    NA
[12,] 0.296 0.279    NA    NA    NA    NA 0.236
[13,]    NA    NA    NA    NA 0.333 0.269    NA
[14,] 0.139    NA    NA 0.272    NA    NA 0.266
[15,]    NA 0.217    NA 0.362    NA    NA 0.292
[16,]    NA    NA 0.178    NA 0.332 0.230    NA
```

```
> print(out$acceptx, digits = 3)
```

```
 [1] 0.1963 0.0969 0.0786 0.0568 0.1057 0.0559 0.0504 0.0361
 [9] 0.1056 0.0667 0.0585 0.0325 0.0618 0.0378 0.0339 0.0243
```

The acceptance rates for swaps seem o. k.

```
> min(as.vector(out$accepti), na.rm = TRUE)
```

```
[1] 0.1394892
```

and there is nothing simple we can do to adjust them (adjustment is possible, see the discussion section for more on this issue). We adjust the acceptance rates for within model moves by adjusting the scaling.

```
> out <- temper(out, scale = 0.5, log.pseudo.prior = qux)
> time.save <- rbind(time.save, out$time, deparse.level = 0)
> print(out$acceptx, digits = 3)
```

```
 [1] 0.400 0.287 0.251 0.222 0.314 0.247 0.225 0.206 0.309 0.226
[11] 0.203 0.168 0.247 0.193 0.196 0.153
```

Looks o. k. now.

Inspection of autocorrelation functions for components of `out$ibatch` (not shown) says batch length needs to be at least 4 times longer. We make it 10 times longer for safety.

Because this run takes longer than CRAN vingettes are supposed to take, we save the results to a file and load the results from this file if it already exists.

```
> lout <- suppressWarnings(try(load("bfst2.rda"), silent = TRUE))
> if (inherits(lout, "try-error")) {
+     out <- temper(out, blen = 10 * out$blen, log.pseudo.prior = qux)
+     save(out, file = "bfst2.rda")
+ } else {
+     .Random.seed <- out$final.seed
+ }
> time.save <- rbind(time.save, out$time, deparse.level = 0)
> foo <- apply(time.save, 2, sum)
> foo.min <- floor(foo[1] / 60)
> foo.sec <- foo[1] - 60 * foo.min
> c(foo.min, foo.sec)

user.self user.self
    5.000    24.612
```

The total time for all runs of the temper function was 5 minutes and 24.6 seconds.

3.3 Bayes Factor Calculations

Now we calculate log 10 Bayes factors relative to the model with the highest unnormalized Bayes factor.

```
> log.10.unnorm.bayes <- (qux - log(colMeans(out$ibatch))) / log(10)
> k <- seq(along = log.10.unnorm.bayes)[log.10.unnorm.bayes
+     == min(log.10.unnorm.bayes)]
> models[k, ]

x1 x2 x3 x4
 1  1  0  1

> log.10.bayes <- log.10.unnorm.bayes - log.10.unnorm.bayes[k]
> log.10.bayes
```

```
 [1] 8.17814103 4.17098637 6.33069128 4.05292216 1.80254545
 [6] 0.67203156 1.40468558 0.70498671 2.58875400 1.93202268
[11] 2.82341431 2.37170521 0.08004553 0.00000000 0.37357715
[16] 0.35242443
```

These are base 10 logarithms of the Bayes factors against the k-th model where $k = 14$. For example, the Bayes factor for the k-th model divided by the Bayes factor for the first model is $10^{8.178}$.

Now we calculate Monte Carlo standard errors two different ways. One is the way the delta method is usually taught. To simplify notation, denote the Bayes factors

$$b_m = b(y \mid m)$$

and their Monte Carlo approximations $\hat{b}_m$. Then the log Bayes factors are

$$g_i(b) = \log_{10} b_i - \log_{10} b_k$$

hence we need to apply the delta method with the function g_i, which has derivatives

$$\begin{aligned}
\frac{\partial g_i(b)}{\partial b_i} &= \frac{1}{b_i \log_e(10)} \\
\frac{\partial g_i(b)}{\partial b_k} &= -\frac{1}{b_k \log_e(10)} \\
\frac{\partial g_i(b)}{\partial b_j} &= 0, \qquad j \neq i \text{ and } j \neq k
\end{aligned}$$

```
> fred <- var(out$ibatch) / out$nbatch
> sally <- colMeans(out$ibatch)
> mcse.log.10.bayes <- (1 / log(10)) * sqrt(diag(fred) / sally^2 -
+     2 * fred[ , k] / (sally * sally[k]) +
+     fred[k, k] / sally[k]^2)
> mcse.log.10.bayes

 [1] 0.02297789 0.02297554 0.02427784 0.02466045 0.02124325
 [6] 0.01925444 0.02381757 0.02289809 0.02190834 0.01983693
[11] 0.02293685 0.02213382 0.01753144 0.00000000 0.02146088
[16] 0.01857454

> foompter <- cbind(models, log.10.bayes, mcse.log.10.bayes)
> round(foompter, 5)
```

```
      x1 x2 x3 x4 log.10.bayes mcse.log.10.bayes
 [1,]  0  0  0  0      8.17814           0.02298
 [2,]  0  0  0  1      4.17099           0.02298
 [3,]  0  0  1  0      6.33069           0.02428
 [4,]  0  0  1  1      4.05292           0.02466
 [5,]  0  1  0  0      1.80255           0.02124
 [6,]  0  1  0  1      0.67203           0.01925
 [7,]  0  1  1  0      1.40469           0.02382
 [8,]  0  1  1  1      0.70499           0.02290
 [9,]  1  0  0  0      2.58875           0.02191
[10,]  1  0  0  1      1.93202           0.01984
[11,]  1  0  1  0      2.82341           0.02294
[12,]  1  0  1  1      2.37171           0.02213
[13,]  1  1  0  0      0.08005           0.01753
[14,]  1  1  0  1      0.00000           0.00000
[15,]  1  1  1  0      0.37358           0.02146
[16,]  1  1  1  1      0.35242           0.01857
```

An alternative calculation of the MCSE replaces the actual function of the raw Bayes factors with its best linear approximation

$$\frac{1}{\log_e(10)} \left(\frac{\hat{b}_i - b_i}{b_i} - \frac{\hat{b}_k - b_k}{b_k} \right)$$

and calculates the standard deviation of this quantity by batch means

```
> ibar <- colMeans(out$ibatch)
> herman <- sweep(out$ibatch, 2, ibar, "/")
> herman <- sweep(herman, 1, herman[ , k], "-")
> mcse.log.10.bayes.too <- (1 / log(10)) *
+     apply(herman, 2, sd) /sqrt(out$nbatch)
> all.equal(mcse.log.10.bayes, mcse.log.10.bayes.too)

[1] TRUE
```

4 Discussion

We hope readers are impressed with the power of this method. The key to the method is pseudopriors adjusted by trial and error. The method could have been invented by any Bayesian who realized that the priors on models,

pri(m) in our notation in Section 1.1, do not affect the Bayes factors and hence are irrelevant to calculating Bayes factors. Thus the priors (or pseudopriors in our terminology) should be chosen for reasons of computational convenience, as we have done, rather than to incorporate prior information.

The rest of the details of the method are unimportant. The `temper` function in R is convenient to use for this purpose, but there is no reason to believe that it provides optimal sampling. Samplers carefully designed for each particular application would undoubtedly do better. Our notion of "padding" so that the within model parameters have the same dimension for all models follows Carlin and Chib (1995) but "reversible jump" samplers (Green, 1995) would undoubtedly do better. Unfortunately, there seems to be no way to code up a function like `temper` that uses "reversible jump" and requires no theoretical work from users that if messed up destroys the algorithm. The `temper` function is foolproof in the sense that if the log unnormalized density function written by the user (like our `ludfun`) is correct, then the ST Markov chain has the equilibrium distribution is supposed to have. There is nothing the user can mess up except this user written function. No analog of this for "reversible jump" chains is apparent (to your humble author).

Two issues remain where the text above said "see the discussion section for more on this issue." The first was about within model priors for the "padding" components of within model parameter vectors $g_{\text{pad}}(\theta_{\text{pad}} \mid m)$ in the notation in (4). Rather than choose these so that they do not depend on the data (as we did), it would be better (if more trouble) to choose them differently for each "padding" component, centering $g_{\text{pad}}(\theta_{\text{pad}} \mid m)$ so the distribution of a component of θ_{pad} is near to the marginal distribution of the same component in neighboring models (according to the `neighbors` argument of the `temper` function).

The other remaining issue is adjusting acceptance rates for jumps. There is no way to adjust this other than by changing the number of models and their definitions. But the models we have cannot be changed; if we are to calculate Bayes factors for them, then we must sample them as they are. But we can insert new models between old models. For example, if the acceptance for swaps between model i and model j is too low, then we can insert distribution k between them that has unnormalized density

$$h_k(x) = \sqrt{h_i(x) h_j(x)}.$$

This idea is inherited from simulated tempering; (Geyer and Thompson, 1995) have much discussion of how to insert additional distributions into a

tempering network. It is another key issue in using tempering to speed up sampling. It is less obvious in the Bayes factor context, but still an available technique if needed.

References

Carlin, B. P. and Chib, S. (1995). Bayesian model choice via Markov chain Monte Carlo methods. *Journal of the Royal Statistical Society, Series B*, **57**, 473–484.

Geyer, C. J. (1994). On the convergence of Monte Carlo maximum likelihood calculations. *Journal of the Royal Statistical Society, Series B*, **56** 261–274.

Geyer., C. J. (2009). *mcmc: Markov Chain Monte Carlo*. R package version 0.7-2, available from CRAN.

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

Green, P. J. (1995). Reversible jump Markov chain Monte Carlo computation and Bayesian model determination. *Biometrika*, **82**, 711–732.

Hoeting, J. A., Madigan, D., Raftery, A. E. and Volinsky, C. T. (1999). Bayesian model averaging: A tutorial (with discussion). *Statical Science*, **19**, 382–417. The version printed in the journal had the equations messed up in the production process; a corrected version is available at `http://www.stat.washington.edu/www/research/online/1999/hoeting.pdf`.

Marinari, E., and Parisi G. (1992). Simulated tempering: A new Monte Carlo Scheme. *Europhysics Letters*, **19**, 451–458.

R Development Core Team (2010). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. `http://www.R-project.org`.

Sung, Y. J. and Geyer, C. J. (2007). Monte Carlo likelihood inference for missing data models. *Annals of Statistics*, **35**, 990–1011.

Thompson, E. A. and Guo, S. W. (1991). Evaluation of likelihood ratios for complex genetic models. *IMA J. Math. Appl. Med. Biol.*, **8**, 149–169.

Torrie, G. M., and Valleau, J. P. (1977). Nonphysical sampling distributions in Monte Carlo free-energy estimation: Umbrella sampling. *Journal of Computational Physics*, **23**, 187–199.