The DNA in Us

by Fabricia Nascimento

Tutorial: Bayesian MCMC phylogenetics using R

Posted on **March 3, 2017**

Bayesian MCMC methods have become incredibly popular in recent times as they allow the implementation of arbitrarily complex models of evolution for the estimation of species phylogenies, species divergence times, and species delimitation under the multi-species coalescent, among many other applications in evolutionary biology. However, many key concepts and issues of MCMC methods appear to be arcane to the average biologist. The purpose of this tutorial is thus to open the MCMC black box, and offer the biologist a step-by-step guide on how to write a phylogenetics MCMC algorithm and explore all the main concepts and issues involved.

This tutorial focuses on writing a Bayesian MCMC algorithm to estimate the molecular distance between two species, d, and the transition-transversion (ts/tv) ratio, k, under Kimura's 1980 nucleotide substitution model. Concepts such as burn-in, convergence and mixing of the MCMC, proposal densities, autocorrelation and effective-sample size are discussed in detail. The reader is assumed to know the very basics of phylogenetic inference with the Bayesian method and the basics of nucleotide substitution models (a good place to start is Ziheng Yang's excellent book "Molecular Evolution: A statistical Approach" published by Oxford University Press). The user is also assumed to have a reasonable command of the R language.

The tutorial is divided into 4 parts:

Part 1: Introduction
Part 2: Markov chain Monte Carlo (MCMC)
Part 3: Efficiency of the MCMC chain
Part 4: Diagnosing the MCMC chain

The complete R script can be downloaded from Github.

https://github.com/thednainus/Bayesian_tutorial

The DNA in Us

by Fabricia Nascimento

Part 1: Introduction

Posted on **March 8, 2017**

Back to Preamble

In this part we introduce the data, and write the appropriate R code to calculate the likelihood, the prior and the (un-normalised) posterior distribution. To avoid numerical problems, we will work with the logarithm of the densities.

The data are a pairwise alignment of the 12S rRNA gene of human and orangutang. The alignment has 948 base pairs (bp) and 90 differences (84 transitions and 6 transversions). See Table 1.3 in p.7 of Yang (2014) "Molecular Evolution: A Statistical Approach" for details.

To represent the data, we use the following R code:

```
n <- 948 # length of alignment in bp
ns <- 84 # total number of transitions (23+30+10+21)
nv <- 6  # total number of transversions (1+0+2+0+2+1+0+0)

V <- nv/n # proportion of transversional differences
S <- ns/n # proportion of transitional differences
```

The log-likelihood function, $\log f(D|d,k)$, for data $D = (n_s, n_v)$, using Kimura's (1980) substitution model [see p.8 in Yang (2014)] is

$$\log f(D|d,k) = (n - n_s - n_v)\log(p_0/4) + n_s \log(p_1/4) + n_v \log(p_2/4),$$

where

$$p_0 = 1/4 + 1/4 \times e^{-4d/(k+2)} + 1/2 \times e^{-2d(k+1)/(k+2)},$$
$$p_1 = 1/4 + 1/4 \times e^{-4d/(k+2)} - 1/2 \times e^{-2d(k+1)/(k+2)},$$
$$p_2 = 1/4 - 1/4 \times e^{-4d/(k+2)}.$$

Note that the likelihood depends only on two parameters, the distance d and the ts/tv ratio k. All the other variables are part of the data. The corresponding R code is

```
# Kimura's (1980) likelihood for two sequences
k80.lnL <- function(d, k, n=948, ns=84, nv=6) {
```

```
 p0 <- .25 + .25 * exp(-4*d/(k+2)) + .5 * exp(-2*d*(k+1)/(k+2))
 p1 <- .25 + .25 * exp(-4*d/(k+2)) - .5 * exp(-2*d*(k+1)/(k+2))
 p2 <- .25 - .25 * exp(-4*d/(k+2))

 return ((n - ns - nv) * log(p0/4) +
        ns * log(p1/4) + nv * log(p2/4))
}
```

The posterior distribution is given by

$$f(d, k|D) = zf(d)f(k)f(D|d, k),$$

where $f(d)$ and $f(k)$ are the marginal priors on d and k, and z is the normalising constant, given by

$$z = 1/\int\int f(d)f(k)f(D|d, k)\,\mathrm{d}d\,\mathrm{d}k.$$

We set the marginal priors to be gamma distributions: $f(d) = \text{Gamma}(d|2, 20)$ and $f(k) = \text{Gamma}(k|2, 0.1)$. The prior mean of d is 2/20 = 0.1, and the prior mean of k is 2/0.1 = 20. The prior densities reflect the biologist's prior information about the model parameters.

The double integral in z cannot be calculated analytically. It may be calculated numerically (for example by using Gaussian quadrature) but this is cumbersome. The number of integrals in z is usually the same as the number of parameters in the model. Thus a model with three parameters will involve a triple integral, and so on. Numerical methods are notoriously slow and inaccurate for calculating integrals with more than two dimensions. In fact, the intractability of the integrals made the Bayesian method unsuitable for practical data analysis for a long time. The development of the MCMC algorithm (in which calculation of z is avoided), together with the increase in power of modern computers, has led to an explosion of practical applications of the Bayesian method during the past two decades.

We will ignore calculation of z here, and will work with the un-normalised posterior instead. The next R function returns the logarithm of the un-normalised posterior, $\log(f(d) \times f(k) \times f(D|d, k))$

```
ulnP <- function(d, k, n=948, ns=84, nv=6,
                 a.d=2, b.d=20, a.k=2, b.k=.1)

 return (dgamma(d, a.d, b.d, log=TRUE) +
        dgamma(k, a.k, b.k, log=TRUE) +
        k80.lnL(d, k, n, ns, nv))
```

We can now plot the likelihood, prior, and posterior surfaces. First we set up a 100 × 100 grid of d and k points over which the surfaces will be plotted:

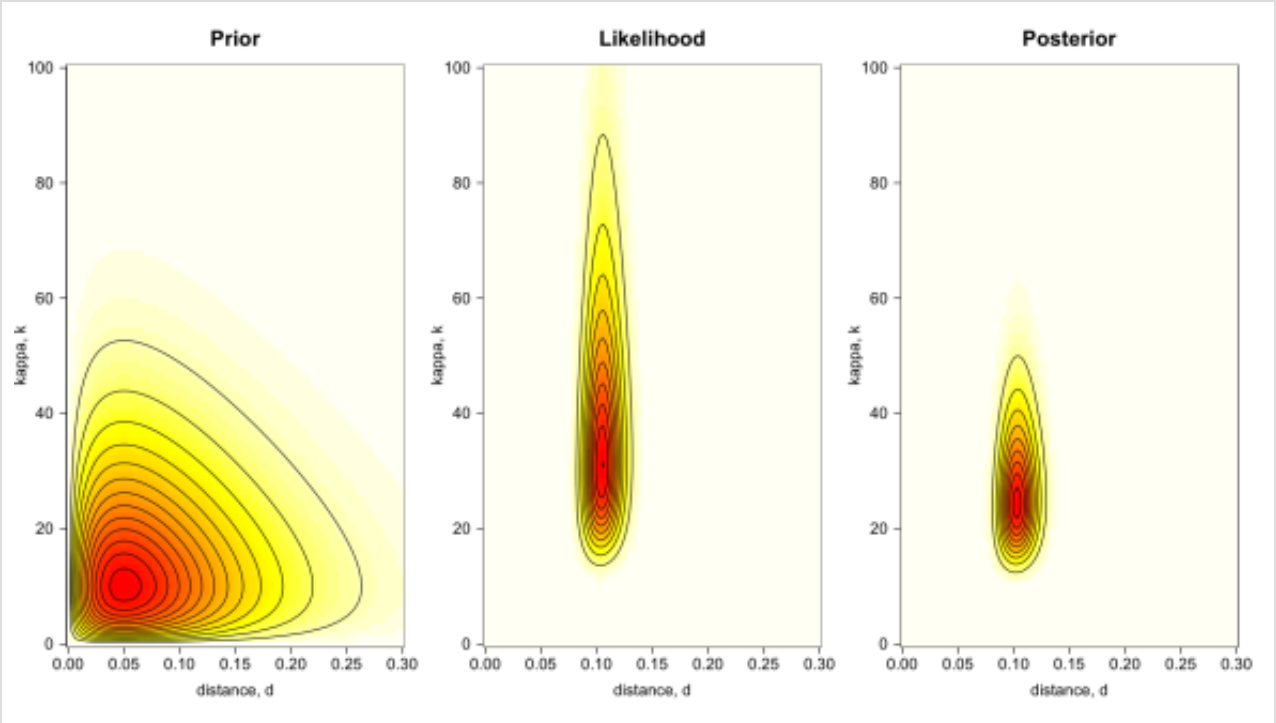
```
dim <- 100  # dimension for the plot
d.v <- seq(from=0, to=0.3, len=dim) # vector of d values
k.v <- seq(from=0, to=100, len=dim) # vector of k values
dk <- expand.grid(d=d.v, k=k.v)

par(mfrow=c(1, 3))
```

We now plot the three surfaces:

```
# Prior surface, f(D)f(k)
Pri <- matrix(dgamma(dk$d, 2, 20) * dgamma(dk$k, 2, .1),
```

```
       ncol=dim)

image(d.v, k.v, -Pri, las=1, col=heat.colors(50),
      main="Prior", xlab="distance, d",
      ylab="kappa, k", cex.main=2.0,
      cex.lab=1.5, cex.axis=1.5)

contour(d.v, k.v, Pri, nlev=10, drawlab=FALSE, add=TRUE)

# Likelihood surface, f(D|d,k)
lnL <- matrix(k80.lnL(d=dk$d, k=dk$k), ncol=dim)

# for numerical reasons, we scale the likelihood to be 1
# at the maximum, i.e. we subtract max(lnL)
L <- exp(lnL - max(lnL))

image(d.v, k.v, -L, las=1, col=heat.colors(50),
      main="Likelihood", xlab="distance, d",
      ylab="kappa, k", cex.main=2.0,
      cex.lab=1.5, cex.axis=1.5)

contour(d.v, k.v, L, nlev=10,
        drawlab=FALSE, add=TRUE) # creates a contour plot

# Unscaled posterior surface, f(d)f(k)f(D|d,k)
Pos <- Pri * L

image(d.v, k.v, -Pos, las=1, col=heat.colors(50),
      main="Posterior", xlab="distance, d",
      ylab="kappa, k", cex.main=2.0,
      cex.lab=1.5, cex.axis=1.5)

contour(d.v, k.v, Pos, nlev=10,
        drawlab=FALSE, add=TRUE)
```

The generated plots are shown below.



In part 2 we show an MCMC algorithm to sample from the posterior surface.

Part 2: Markov chain Monte Carlo (MCMC)

The DNA in Us

by Fabricia Nascimento

Part 2: Markov chain Monte Carlo (MCMC)

Posted on **March 8, 2017**

Back to Part 1

We now aim to obtain the posterior distribution of d and k by MCMC sampling.

The draft MCMC algorithm is:

1. Set initial states for d and k.
2. Propose a new state d^* (from an appropriate proposal density).
3. Accept or reject the proposal with probability $min(1, p(d^*)p(k)p(D|d^*)/p(d)p(k)p(D|d))$
   If the proposal is accepted set $d = d^*$, otherwise $d = d$.
4. Save d.
5. Repeat 2-4 for k.
6. Go to step 2.

The corresponding R function for the MCMC algorithm is:

```r
mcmcf <- function(init.d, init.k, N, w.d, w.k) {
  # init.d and init.k are the initial states
  # N is the number of 'generations' the algorithm is run for.
  # w.d is the 'width' of the proposal density for d.
  # w.k is the 'width' of the proposal density for k.

  # keep the visited values of d and k.
  d <- k <- numeric(N+1)
  d[1] <- init.d
  k[1] <- init.k
  acc.d <- acc.k <- 0 # number of acceptances

  for (i in 1:N) {
    # we use uniform densities with reflection
    # to propose new d* and k* states

    # propose and accept/reject new d
    d.prop <- abs(d[i] + runif(1, -w.d/2, w.d/2))
    lnalpha <- ulnP(d.prop, k[i]) - ulnP(d[i], k[i])
    # if ru < alpha accept proposed d*:     if (lnalpha > 0 || runif(1, 0, 1) < exp
      d[i+1] <- d.prop; acc.d  <- acc.d + 1
    }
    # else reject it:
```

```
      else d[i+1] <- d[i]

      # propose and accept/reject new k
      k.prop <- abs(k[i] + runif(1, -w.k/2, w.k/2))
      lnalpha <- ulnP(d[i], k.prop) - ulnP(d[i], k[i])
      # if ru < alpha accept proposed k*:     if (lnalpha > 0 || runif(1, 0, 1) < exp
        k[i+1] <- k.prop; acc.k  <- acc.k + 1
      }
      # else reject it:
      else k[i+1] <- k[i]
    }

    # print out the proportion of times
    # the proposals were accepted
    print("Acceptance proportions (d, k):")
    print(c(acc.d/N, acc.k/N))

    # return vector of d and k visited during MCMC
    return (list(d=d, k=k))
  }
```

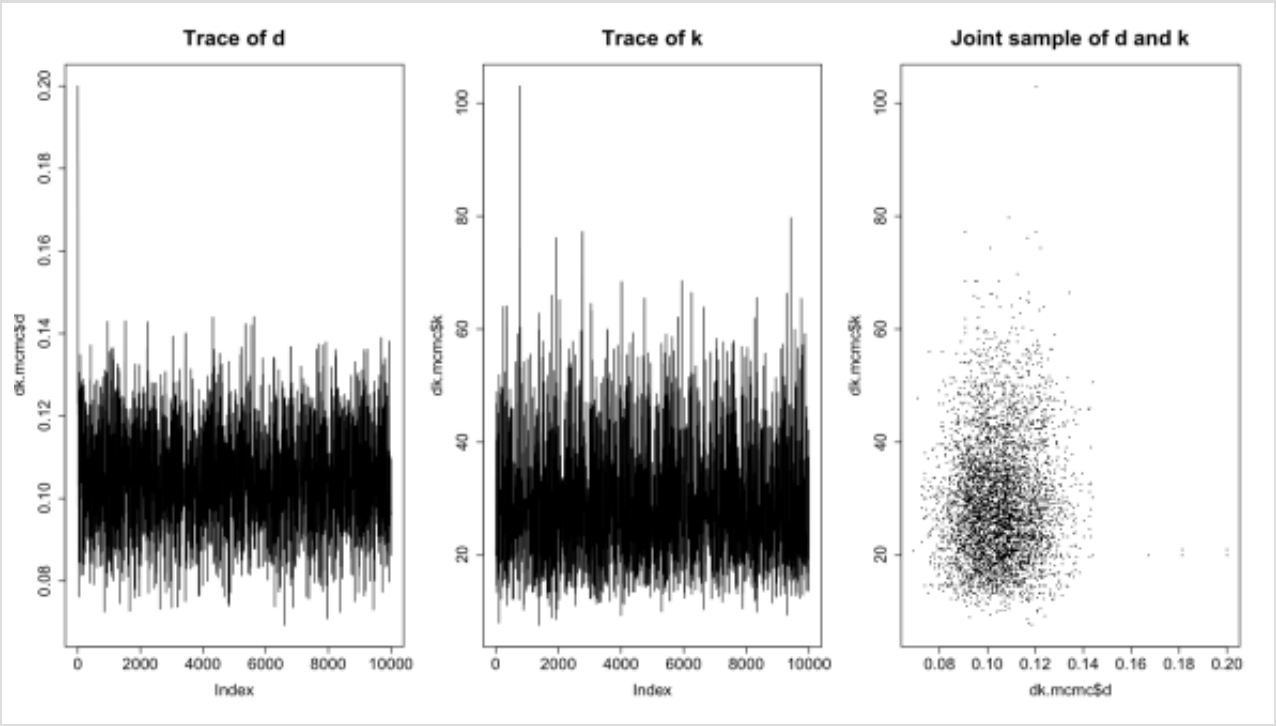
We now test the MCMC algorithm. Note that the algorithm is random, so repetitions of it will give different results. Function `system.time` in R is useful to measure the amount of time an R expression takes to be evaluated. For example, running the MCMC algorithm above with 10,000 iterations takes about 0.7 seconds in a 2.2 GHz MacBook Air.

```
# Test run-time:
system.time(mcmcf(0.2, 20, 1e4, .12, 180)) # about 0.7s
# Run again and save MCMC output:
dk.mcmc <- mcmcf(0.2, 20, 1e4, .12, 180)
```

We now plot the 'traces' of the parameters. A trace plot is a plot of parameter state vs. MCMC iteration. Trace plots allow us to easily visualise the MCMC chain. They can also show us if there are problems with the chain (we'll see this in Part 3):

```
par(mfrow=c(1,3))

# trace plot of d
plot(dk.mcmc$d, ty='l', xlab="Iteration", ylab="d", main="Trace of d")
# trace plot of k
plot(dk.mcmc$k, ty='l', xlab="Iteration", ylab="k", main="Trace of k")

# We can also plot the joint sample of d and k
# (points sampled from posterior surface)
plot(dk.mcmc$d, dk.mcmc$k, pch='.', xlab="d", ylab="k", main="Joint of d and k")
```

Trace plots for distance, d and kappa, k, and the plot for joint sampled from posterior surface.



The trace plots show that the chain has mixed well (i.e., it has explored the parameter space well). The joint sample of d and k matches the plot of the posterior surface from Part 1. In Part 3 we will see examples of chains that have poor mixing. Chains with poor mixing may be the product of poorly chosen proposal densities of bad proposal step sizes. For example, trying running the MCMC chain again after modifying the step sizes (the widths) of the proposal densities, `w.d` and `w.k`.

Part 3: Efficiency of the MCMC chain

The DNA in Us

by Fabricia Nascimento

Part 3: Efficiency of the MCMC chain

Posted on **March 8, 2017**

Back to Part 2

Values sampled in an MCMC chain are autocorrelated because new states are either the previous state or a modification of it. The efficiency of an MCMC chain is closely related to the autocorrelation. Intuitively, if the autocorrelation is high, the chain will be inefficient, i.e. we will need to run the chain for a long time to obtain a good approximation to the posterior distribution.

The efficiency of a chain is defined as

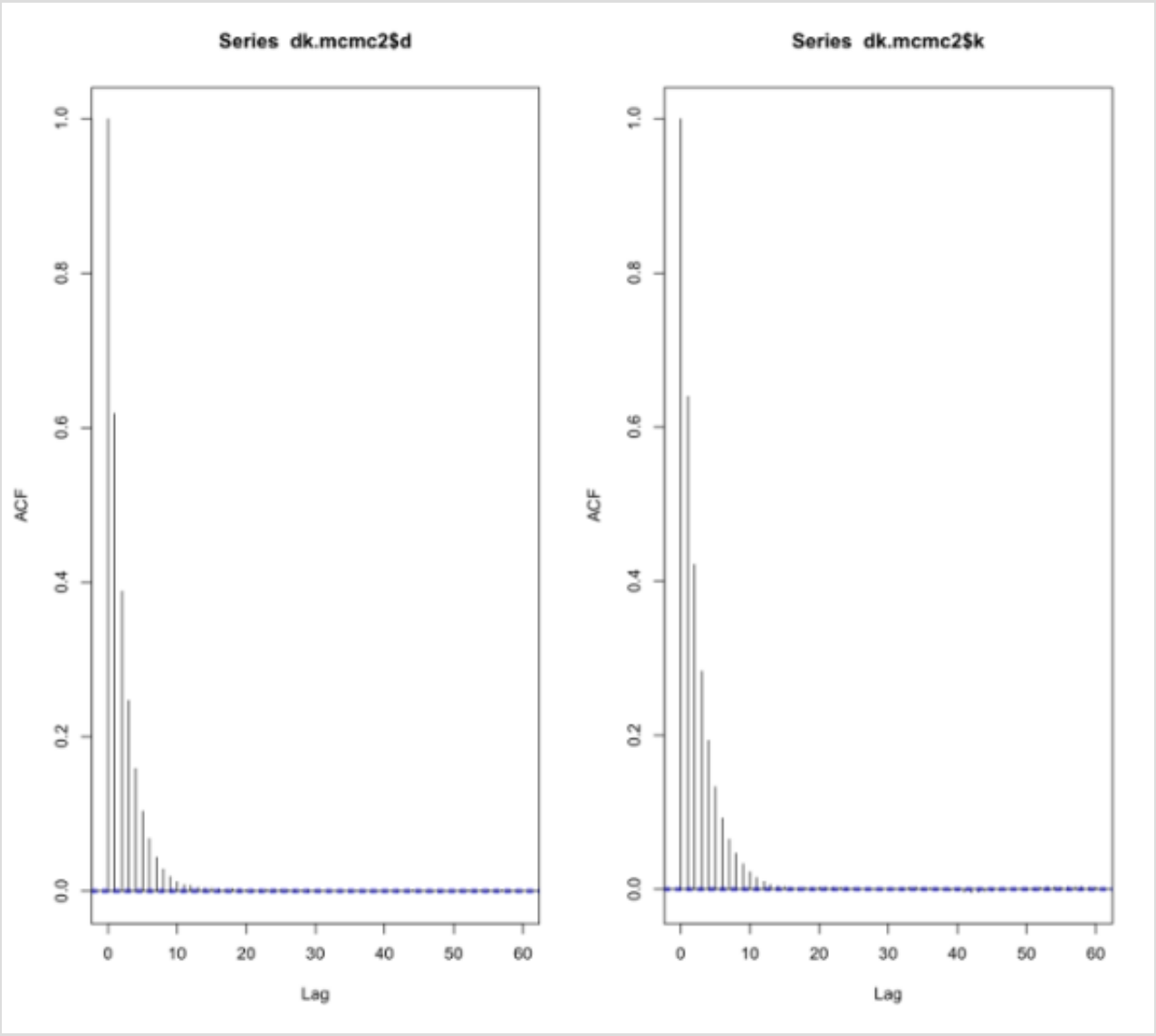
$$\text{eff} = 1/(1 + 2(r_1 + r_2 + r_3 + ...)),$$

where r_i is the correlation coefficient for lag i. We will now calculate the efficiency of the MCMC. To calculate the efficiency it is appropriate to obtain a long sample (say one million). R's autocorrelation function, `acf`, can be used to calculate the correlation coefficients and plot them in the so-called autocorrelation plot.

```
# run a very long chain (1e6 generations take about
# 1.2min in my MacBook Air) to calculate efficiency
dk.mcmc2 <- mcmcf(0.2, 20, 1e6, .12, 180)

# R's acf function (for AutoCorrelation Function)
par(mfrow=c(1,2))
acf(dk.mcmc2$d)
acf(dk.mcmc2$k)

# Define efficiency function
eff <- function(acf) 1 / (1 + 2 * sum(acf$acf[-1]))

# the efficiencies are roughly 22% and 20% for d and k respectively:
eff(acf(dk.mcmc2$d)) # [1] 0.2255753 # mcmcf(0.2, 20, 1e7, .12, 180)
eff(acf(dk.mcmc2$k)) # [1] 0.2015054 # mcmcf(0.2, 20, 1e7, .12, 180)
```

The chain has an efficiency of roughly 20%-22% for k and d. This means that the chain is about as 20% as efficient as independent sampling.

The plots for the autocorrelation are shown below:



Notice that the autocorrelation drops to zero after about 10-15 iterations for both d and k. That is, a sample observation obtained at iteration i, is roughly uncorrelated to an observation obtained at iteration $i + 15$ (i.e. for a lag of 15).

Also notice that the autocorrelation is very high for small lags. For example, for a lag of 2, the autocorrelation is roughly 60%. This is because neighbouring sates in the chain are highly similar. It is customary to 'thin' the chain, that is, save states only every other iteration or so. This is done to remove the highly correlated neighbouring states, which can be done without loosing too much information, and thus save hard drive space in the computer. However the estimates with the smallest variance (error) are obtained when all of the samples in the chain are used.

A concept closely related to efficiency is the effective-sample size (ESS) of the chain. It is defined as

$$\text{ESS} = N * \text{eff},$$

where N is the total number of samples taken from the chain. Thus for the chain above, the ESS's are

```
1e6 * 0.23 # ~230K
1e6 * 0.20 # ~200K
```

or about 200,000s. This means that an independent sample of size 200K has about the same variance (error) for

the parameter estimates (in this case the mean of d and k) than our autocorrelated MCMC chain of size 1 million. An MCMC chain with efficiency around 20% is a very efficient chain. We will see examples of inefficient chains next.

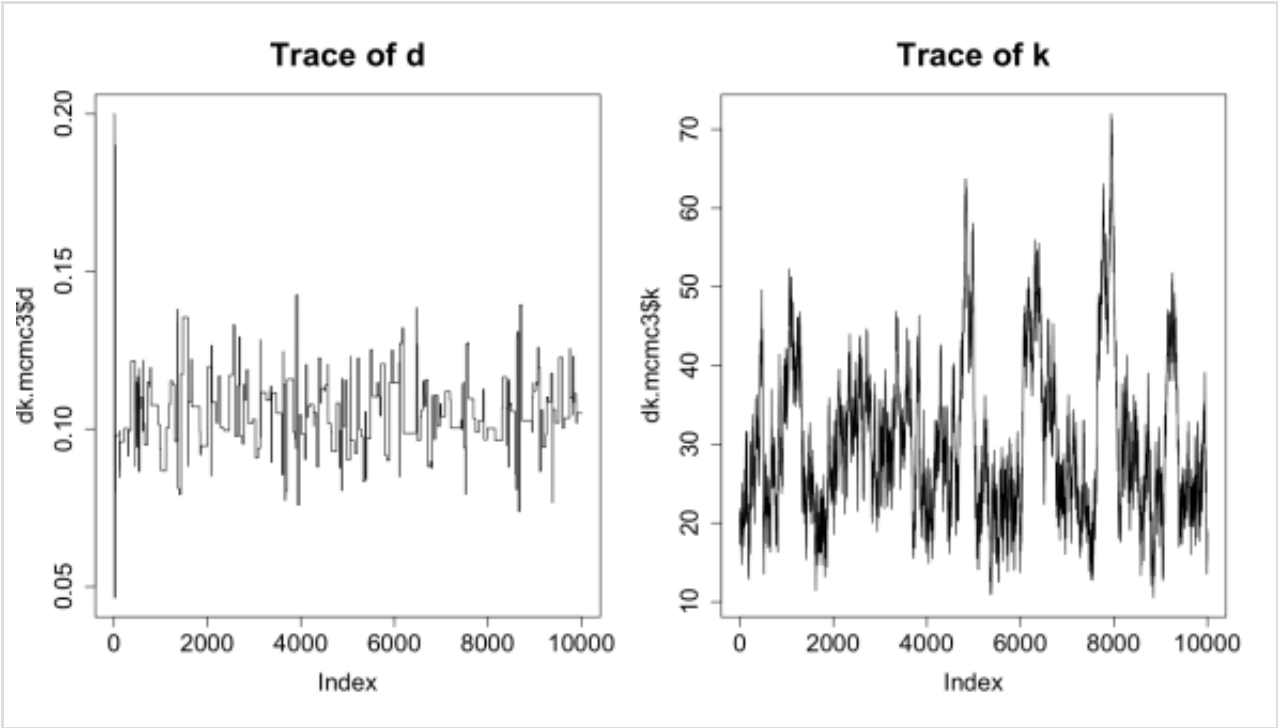
Inefficient chains:

Inefficient chains are those that do not 'mix' well. That is, parameter states are highly autocorrelated even when they are many iterations apart, and thus both the efficiency and the ESS tend to be low. Inefficient chains usually occur due to poor proposal densities, or when sampling from complex posteriors, such as highly correlated posteriors in some over-parameterised models.

To illustrate inefficient chains, we will run our MCMC again by using a proposal density with a too large step size for d, and another with a too small step size for k.

```
# The window width for the d proposal density is too large,
# while it is too small for k
dk.mcmc3 <- mcmcf(0.2, 20, 1e4, 3, 5)

par(mfrow=c(1,2))
# because proposal width for d is too large,
# chain gets stuck at same values of d:
plot(dk.mcmc3$d, ty='l', main="Trace of d", cex.main=2.0,
     cex.lab=1.5, cex.axis=1.5, ylab="d")

# whereas proposal width for k is too small,
# so chain moves slowly:
plot(dk.mcmc3$k, ty='l', main="Trace of k", cex.main=2.0,
     cex.lab=1.5, cex.axis=1.5, ylab="k")
```

The trace plots for the inefficient chain are shown below. In the case of d, the proposal density is too wide, a new proposals of d are so far from the current state that they may fall outside the area of high probability density in the posterior, and thus most proposals are rejected. This is evident in the shape of the trace plot, which has many horizontal sections, i.e., portions where the chain remained stuck in the same value of d for many iterations. In the case of k the proposal density is actually too narrow, and so the chain moves slowly by making baby-sized steps.

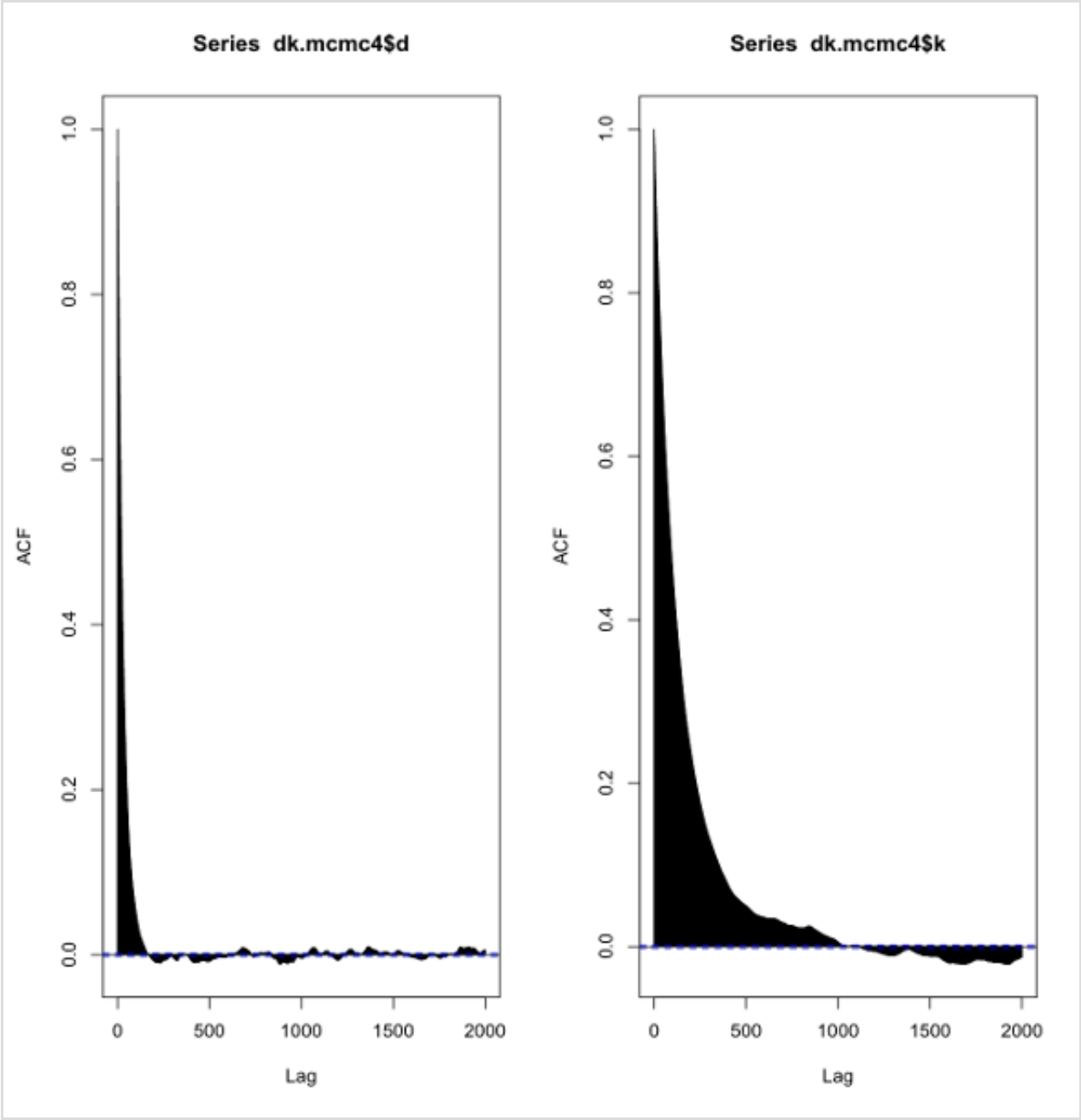


Inefficient chains are characterised by acceptance proportions that are either too low or too high. For d the acceptance proportion (check your R output) was about ~3% (i.e. over 97% of proposals were rejected), while for k the acceptance proportion was over 90% (i.e. the vast majority of proposals were accepted). As a rule of thumb, for bell-shaped posterior densities, one should aim for acceptance proportions close to 30%, which lead to optimal mixing, large efficiency and large ESS. Try tweaking the values of `w.d` and `w.k` and running the MCMC again until you achieve acceptance proportions close to 30% for both parameters. Then check the trace plots. Finding the optimal step sizes is known as 'fine-tuning' the chain. Many MCMC programs use automatic fine-tuning algorithms and the user does not normally need to worry about this.

We will now calculate `eff` for the inefficient chains. Because the chain has so much autocorrelation, it is best to do a very long run (using 1e7 iterations is best, but this takes over 10min in my computer).

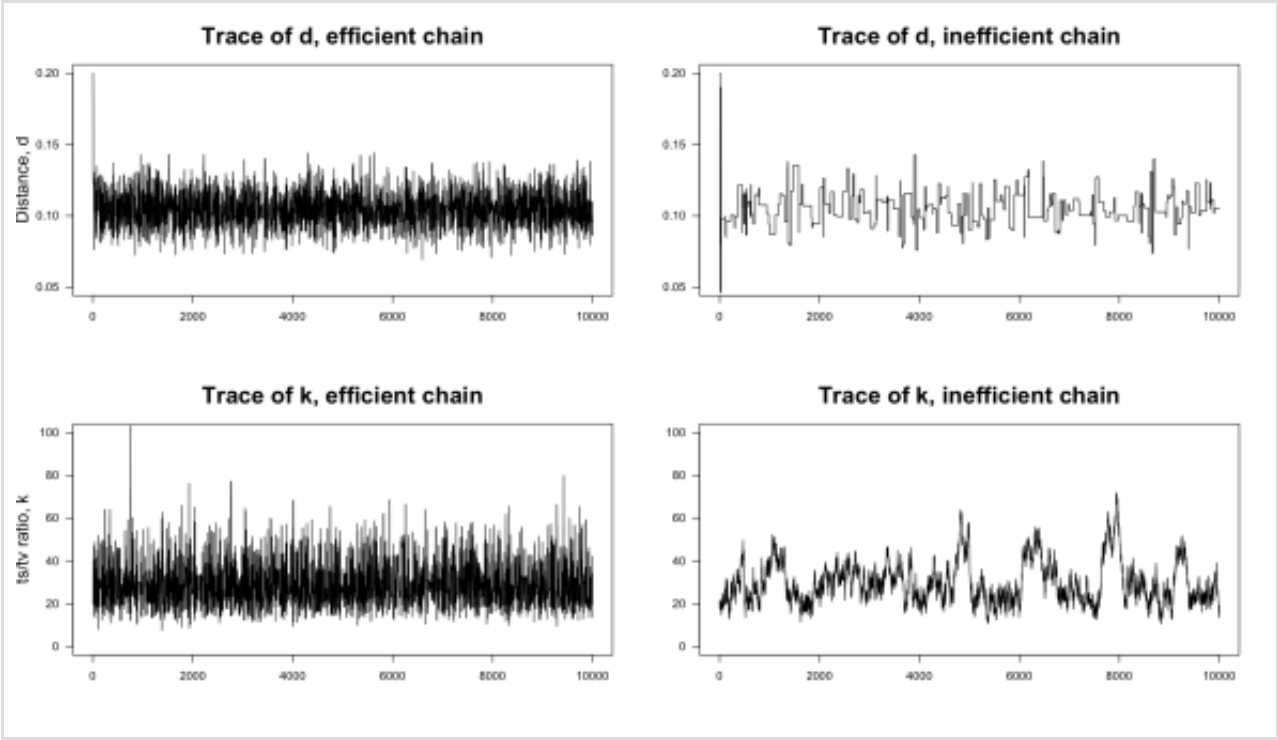
```
dk.mcmc4 <- mcmcf(0.2, 20, 1e6, 3, 5)

# Efficiencies are roughly 1.5% for d, and 0.35% for k:
eff(acf(dk.mcmc4$d, lag.max=2e3)) # [1] 0.01530385  # mcmcf(0.2, 20, 1e7, 3, 5)
eff(acf(dk.mcmc4$k, lag.max=2e3)) # [1] 0.003493112 # mcmcf(0.2, 20, 1e7, 3, 5)
```

The efficiencies are very slow, about 1.5% and 0.3% for d and k respectively. This means ESS's around 15K and 3K for a chain of size 1 million. This is very inefficient. The autocorrelation plots are shown below. Notice that the correlations only drop to zero for about 200 iterations in the case of d and over 1,000 iterations in the case of k.



Finally, we plot the traces for efficient (part 2) and inefficient chains.

```
par(mfrow=c(2,2))

plot(dk.mcmc$d, ty='l', las=1, ylim=c(.05,.2),
     main="Trace of d, efficient chain", xlab='',
     ylab="Distance, d", cex.main=2.0, cex.lab=1.5)
plot(dk.mcmc3$d, ty='l', las=1, ylim=c(.05,.2),
     main="Trace of d, inefficient chain", xlab='',
     ylab='', cex.main=2.0, cex.lab=1.5)
plot(dk.mcmc$k, ty='l', las=1, ylim=c(0,100),
     main="Trace of k, efficient chain",
     xlab='', ylab="ts/tv ratio, k",
     cex.main=2.0, cex.lab=1.5)
plot(dk.mcmc3$k, ty='l', las=1, ylim=c(0,100),
     main="Trace of k, inefficient chain",
     xlab='', ylab='', cex.main=2.0, cex.lab=1.5)
```



In this part we explored efficiency, effective-sample size and acceptance proportions. These concepts are important when assessing whether the chain is mixing well or not. However, chains that may appear to mix well may not actually be converging to the posterior distribution (for example, if the posterior is strongly multi-modal). Thus, efficiency and ESS may tell us little (if nothing) about whether the chain has converged. We look at this in the next section.

Part 4: Checking for convergence

The DNA in Us

by Fabricia Nascimento

Part 4: Checking for convergence

Posted on **March 8, 2017**

Back to Part 3

The simplest way to check for convergence is to run the MCMC chain several times (at least twice) with over-dispersed starting values. Then one can compare the outputs of the various chains and check how similar they are. If the chains have converged to the target posterior distribution, then they will have similar (almost identical) histograms, posterior means, 95% credibility intervals, etc. The potential scale reduction statistic of Gelman and Rubin can also be calculated and used to assess for convergence.

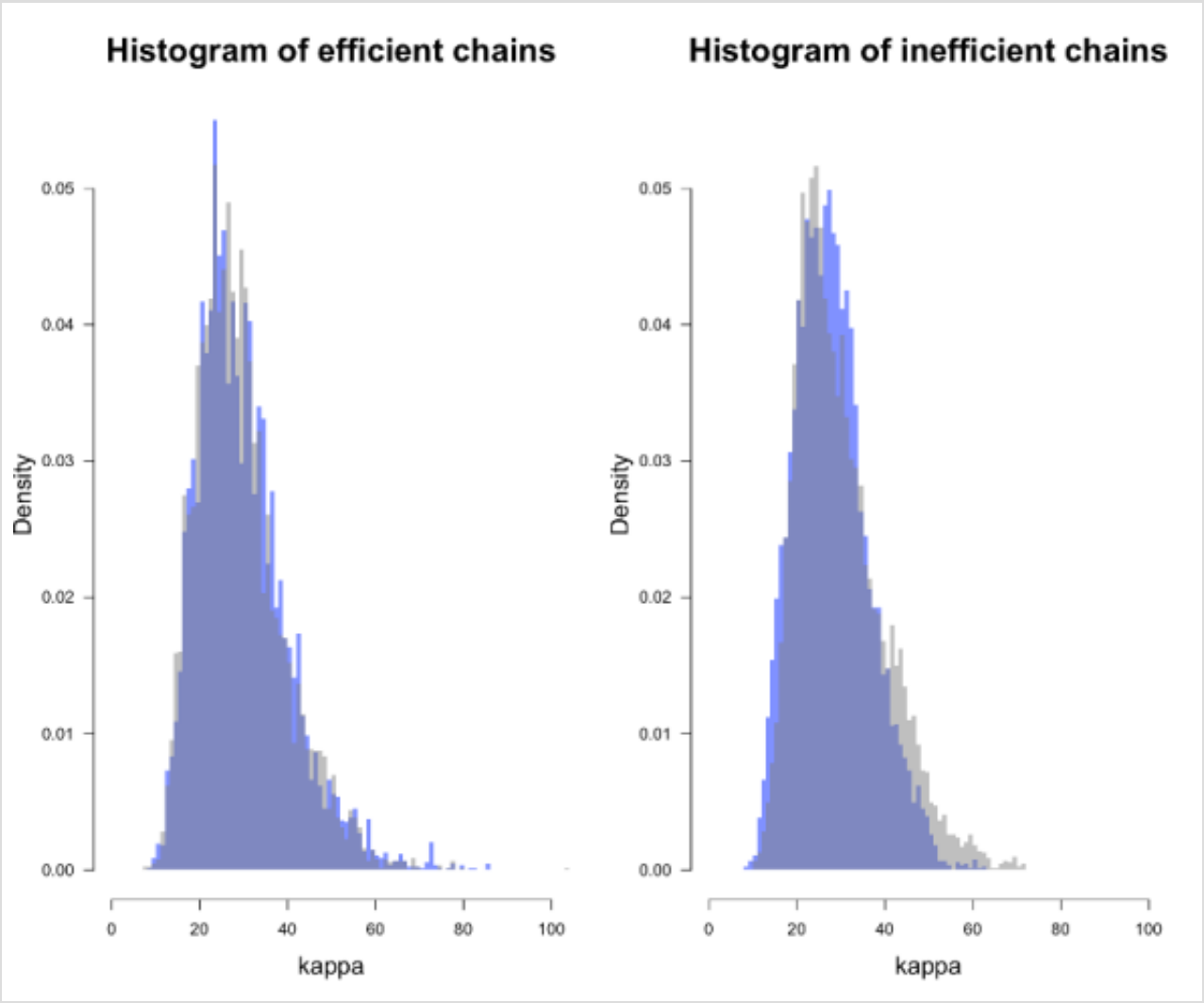
We now run the efficient and inefficient chains again, but with different starting values (i.e. `d.init` and `k.init`)

```
# Efficient chain (good proposal step sizes)
dk.mcmc.b <- mcmcf(0.05, 5, 1e4, .12, 180)
# Inefficient chain (bad proposal step sizes)
dk.mcmc3.b <- mcmcf(0.05, 5, 1e4, 3, 5)
```

And we can now plot and compare the histograms:

```
par(mfrow=c(1,2))
bks <- seq(from=0, to=105, by=1)

hist(dk.mcmc.b$k, prob=TRUE, breaks=bks, border=NA,
     col=rgb(0, 0, 1, .5), las=1, xlab="kappa",
     xlim=c(0,100), ylim=c(0,.055))

hist(dk.mcmc$k, prob=TRUE, breaks=bks, border=NA,
     col=rgb(.5, .5, .5, .5), add=TRUE)

hist(dk.mcmc3.b$k, prob=TRUE, breaks=bks, border=NA,
     col=rgb(0, 0, 1, .5), las=1, xlab="kappa",
     xlim=c(0,100), ylim=c(0,.055))

hist(dk.mcmc3$k, prob=TRUE, breaks=bks, border=NA,
     col=rgb(.5, .5, .5, .5), add=TRUE)
```

Efficient chains converge quickly, and histograms overlap well. On the other hand, inefficient chains need to be run for a longer time to converge, here they were not run long enough, so that the histograms do not overlap as well. Your histograms may look slightly different due to the stochastic nature of the MCMC algorithm.



We can also calculate the posterior means:

```
# posterior means (similar for efficient chains):
mean(dk.mcmc$d); mean(dk.mcmc.b$d)
mean(dk.mcmc$k); mean(dk.mcmc.b$k)

# posterior means (not so similar for the inefficient chains):
mean(dk.mcmc3$d); mean(dk.mcmc3.b$d)
mean(dk.mcmc3$k); mean(dk.mcmc3.b$k)
```

The variance (error) of the estimated posterior mean is the variance of the chain divided by the efficiency times 1/(sample size). The standard error of the estimated posterior mean is the square-root of the variance:

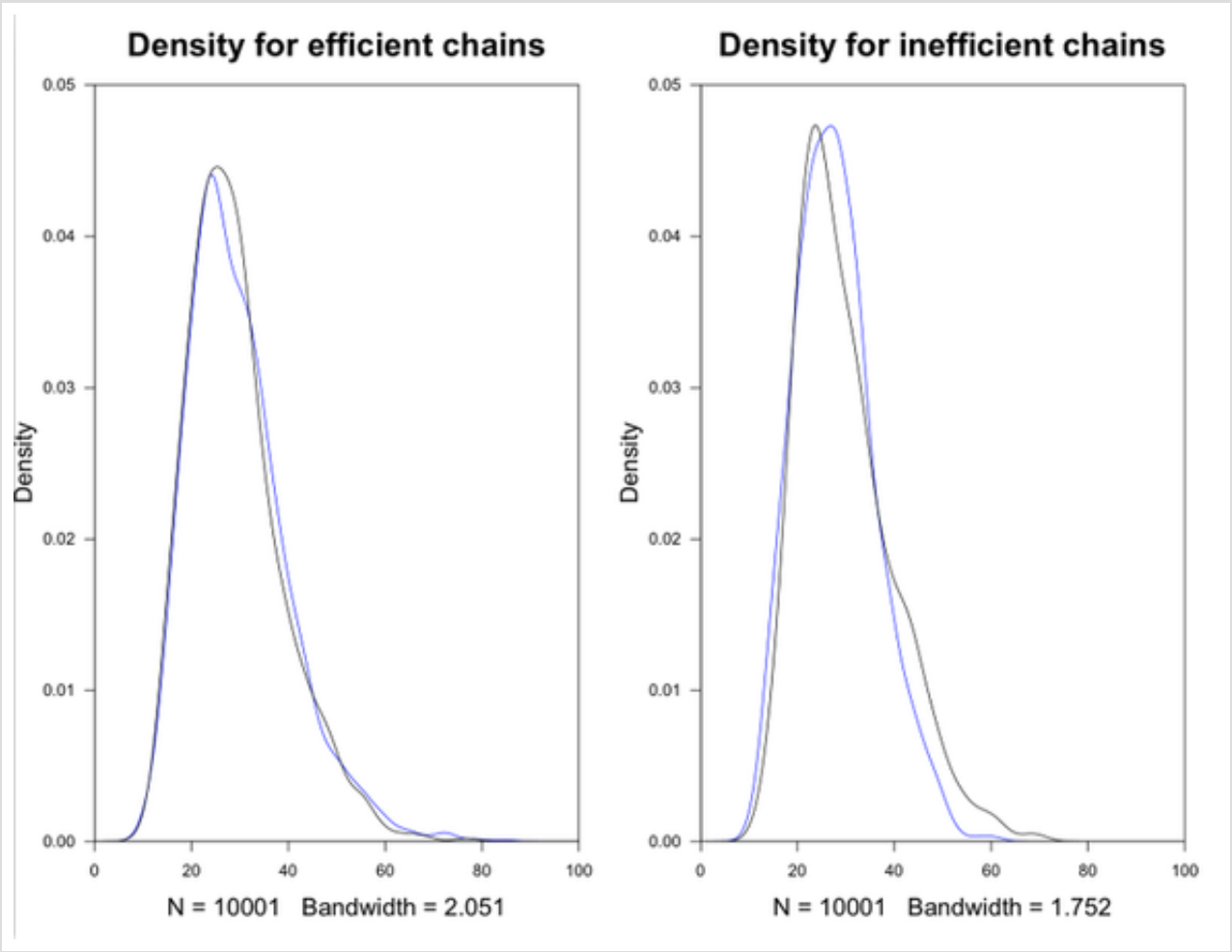
```
# efficient chain, standard error of the means
sqrt(1/1e4 * var(dk.mcmc$d) / 0.23) # roughly 2.5e-4
sqrt(1/1e4 * var(dk.mcmc$k) / 0.20) # roughly 0.2

# inefficient chain, standard error of the means
sqrt(1/1e4 * var(dk.mcmc3$d) / 0.015) # roughly 9.7e-4
sqrt(1/1e4 * var(dk.mcmc3$k) / 0.003) # roughly 1.6
```

Thus, for the efficient chain on d, I got an estimated posterior mean of 0.105, which is within ±2 times 2.5e-4 of the true posterior mean. Note for the inefficient chains, the standard errors are about four times and eight times those of the efficient chains.

We can also plot densities (smoothed histograms) for the efficient and inefficient chains:

```r
par(mfrow=c(1,2)); adj <- 1.5
# Efficient chains:
plot(density(dk.mcmc.b$k, adj=adj), col="blue", las=1,
     xlim=c(0, 100), ylim=c(0, .05), xaxs="i", yaxs="i")
lines(density(dk.mcmc$k, adj=adj), col="black")

# Inefficient chains:
plot(density(dk.mcmc3.b$k, adj=adj), col="blue", las=1,
     xlim=c(0, 100), ylim=c(0, .05), xaxs="i", yaxs="i")
lines(density(dk.mcmc3$k, adj=adj), col="black")
```

The resulting R plots are shown below.



A useful measure of convergence is the potential scale reduction statistic of Gelman and Rubin (also known as Gelman and Rubin convergence diagnostic). The statistic compares the within-chain variance and the among-chain variance for the posterior mean of a parameter. When the value of the statistic is close to 1, convergence has been achieved. Note that the diagnostic may fail (i.e. give a misleading reassurance of convergence) for strongly multi-modal posteriors. To avoid this it is important to run more than two chains with over-dispersed starting values. The R package `coda` for MCMC diagnostics, has a function to calculate the convergence diagnostic, `gelman.diag`. If you have the package installed, you may want to go ahead and calculate the diagnostic for the chains above.

Chapter 7 in Yang (2014, Molecular Evolution: A Statistical Approach) provides a good introduction to Bayesian computation including Bayesian MCMC, proposal densities, efficiency, convergence diagnostics, etc.

We hope you enjoyed the tutorial. If you have questions, comments or if you spot any errors, please post a message below or sends us an email!