AMS 206: Applied Bayesian Statistics

David Draper

Department of Applied Mathematics and Statistics
University of California, Santa Cruz

draper@ucsc.edu
www.ams.ucsc.edu/∼draper

Lecture Notes (Part 4)

3.1 Markov Chain Monte Carlo (MCMC) Methods

Computation via conjugate analysis (Chapter 2) produces **closed-form results** (good) but is **limited in scope** to a fairly small set of models for which straightforward conjugate results are possible (bad); for example, there is **no conjugate prior** for (μ, σ^2, ν) in the NB10 t model.

This was a **severe limitation** for Bayesians for almost 250 years (from the 1750s to the 1980s).

Over the past 25 years or so the Bayesian community has "discovered" and developed an entirely new computing method, **Markov chain Monte Carlo (MCMC)** ("discovered" because the physicists first figured it out about 70 years ago: Metropolis and Ulam, 1949; Metropolis et al., 1953).

It became clear above that the **central Bayesian practical challenge** is the **computation of high-dimensional integrals**.

People working on the first atom bomb in World War II faced a **similar challenge**, and noticed that **digital computers** (which were then passing from theory (Turing 1943) to reality) offered an **entirely new approach** to solving the problem.

Simulation-Based Computation

The idea (Metropolis and Ulam, 1949) was based on the observation that **anything I want to know about a probability distribution can be learned to arbitrary accuracy by sampling from it**.

Suppose, for example, that I'm interested in a posterior distribution $p(\theta|y)$ that **cannot be worked with (easily) in closed form**, and initially (to keep things simple) think of θ as a **scalar** (real number) rather than a vector.

Three things of direct interest to me about $p(\theta|y)$ would be

- its low-order moments, including the **mean** $\mu = E(\theta|y)$ and **standard deviation** $\sigma = \sqrt{V(\theta|y)}$,
- its **shape** (basically I'd like to be able to trace out (an estimate of) the entire **density curve**), and
- one or more of its **quantiles** (e.g., to construct a 95% central posterior interval for θ I need to know the **2.5% and 97.5% quantiles**, and sometimes the **posterior median** (the **50th percentile**) is of interest too).

Simulation-Based Computation (continued)

Suppose I could take an **arbitrarily large random sample** from $p(\theta|y)$, say $\theta^*_1, \dots, \theta^*_m$.

Then each of the above three aspects of $p(\theta|y)$ can be **estimated** from the θ^* sample:

- $\hat{E}(\theta|y) = \bar{\theta}^* = \frac{1}{m} \sum_{j=1}^m \theta^*_j$, and $\sqrt{\hat{V}(\theta|y)} = \sqrt{\frac{1}{m-1} \sum_{j=1}^m \left(\theta^*_j - \bar{\theta}^* \right)^2}$;
- the density curve can be estimated by a **histogram** or **kernel density estimate**; and
- percentiles can be estimated by **counting** how many of the θ^* values fall below a series of specified points — e.g., to find an estimate of the 2.5% quantile I solve the equation

$$\hat{F}_\theta(t) = \frac{1}{m} \sum_{j=1}^m I(\theta^*_j \le t) = 0.025 \qquad (1)$$

for t, where $I(A)$ is the **indicator function** (1 if A is true, otherwise 0).

These are called **Monte Carlo** estimates of the true summaries of $p(\theta|y)$ (in

3.2 IID Sampling; Rejection Sampling

honor of the casinos) because they're based on the **controlled use of chance**.

Theory shows that with large enough m, each of the Monte Carlo (or **simulation-based**) estimates can be made arbitrarily close to the truth with arbitrarily high probability, under some reasonable assumptions about the **nature of the random sampling**.

One way to achieve this, of course, is to make the sampling **IID** (interestingly, this is **sufficient** but **not necessary** — see below).

If, for example, $\bar{\theta}^* = \frac{1}{m} \sum_{j=1}^m \theta_j^*$ is based on an IID sample of size m from $p(\theta|y)$, I can use the **frequentist fact** that in repeated sampling $V\left(\bar{\theta}^*\right) = \frac{\sigma^2}{m}$, where (as above) σ^2 is the variance of $p(\theta|y)$, to construct a **Monte Carlo standard error** (MCSE) for $\bar{\theta}^*$:

$$\widehat{SE}\left(\bar{\theta}^*\right) = \frac{\hat{\sigma}}{\sqrt{m}}, \tag{2}$$

where $\hat{\sigma}$ is the **sample SD** of the θ^* values.

This can be used, possibly after some **preliminary experimentation**, to decide on m, the Monte Carlo **sample size**, which later will be called the

An IID Example

length of the **monitoring run**.

An IID example. Consider the posterior distribution $p(\theta|y) = \text{Beta}(76.5, 353.5)$ in the **AMI mortality example** in Part 2.

Theory says that the **posterior mean** of θ in this example is $\frac{76.5}{76.5+353.5} \doteq 0.1779$; let's see how well the Monte Carlo method does in estimating this **known truth**.

Here's an `R` function to construct **Monte Carlo estimates** of the **posterior mean** and **MCSE values** for these estimates.

```
beta.sim <- function( m, alpha, beta, n.sim, seed ) {
  set.seed( seed )
  theta.out <- matrix( 0, n.sim, 2 )
  for ( i in 1:n.sim ) {
    theta.sample <- rbeta( m, alpha, beta )
    theta.out[ i, 1 ] <- mean( theta.sample )
    theta.out[ i, 2 ] <- sqrt( var( theta.sample ) / m )
 }
```

IID Example (continued)

```
  return( theta.out )
}
```

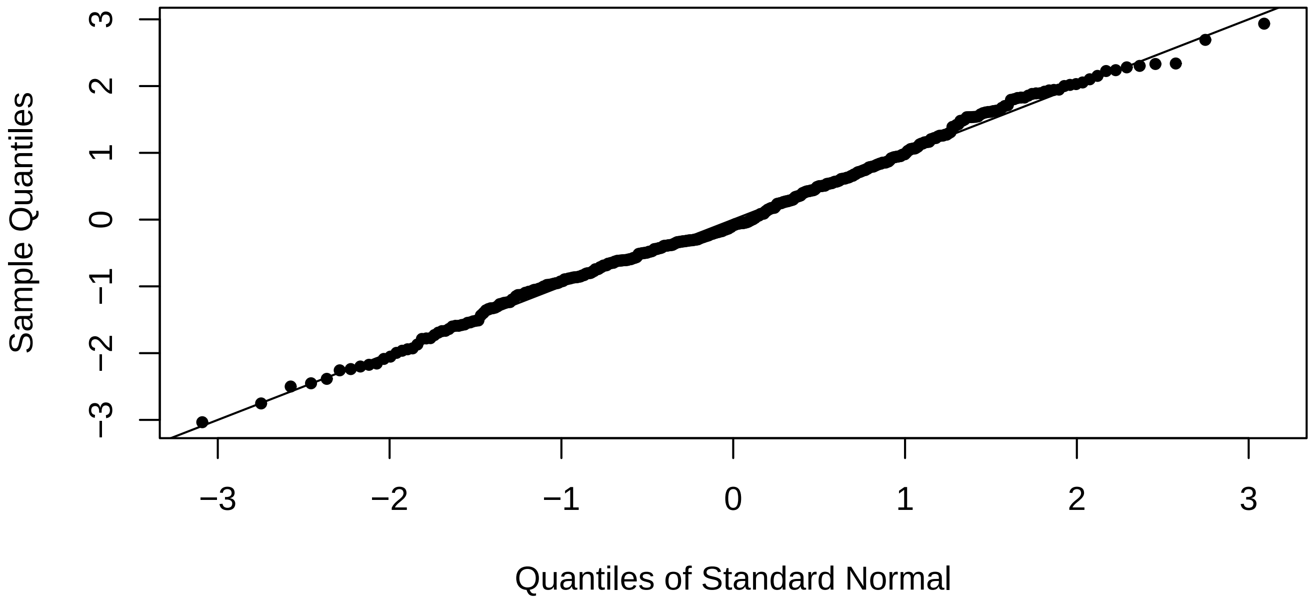
This function simulates, **`n.sim`** times, the process of taking an **IID sample** of size m from the Beta(α, β) distribution and **calculating** $\bar{\theta}^*$ and $\widehat{SE}\left(\bar{\theta}^*\right)$.

```
> m <- 100
> alpha <- 76.5
> beta <- 353.5
> n.sim <- 500
> seed <- c( 6425451, 9626954 )
> theta.out <- beta.sim( m, alpha, beta, n.sim, seed )
# This took about 0.2 second at 1.6 Unix GHz.
> theta.out[ 1:5, ]
           [,1]        [,2]
 [1,] 0.1756400 0.001854220
 [2,] 0.1764806 0.001703780
 [3,] 0.1781742 0.001979863
 [4,] 0.1793588 0.002038532
 [5,] 0.1781556 0.001596011
```

IID Example (continued)

The $\bar{\theta}^*$ values fluctuate around the truth with a **give-or-take** of about 0.0018, which agrees well with the **theoretical SE** $\frac{\sigma}{\sqrt{m}} = \frac{0.0184}{\sqrt{100}} \doteq 0.00184$ (the SD value 0.0184 comes from page 47 in Part 2).

```
> theta.bar <- theta.out[ , 1 ]
> qqnorm( ( theta.bar - mean( theta.bar ) ) / sd( theta.bar ),
    xlab = "Quantiles of Standard Normal", main = "", pch = 20 )
> abline( 0, 1 )
```



IID Example (continued)

Each of the $\bar{\theta}^*$ values is the mean of $m = 100$ IID draws, so (by the CLT) the **distribution** of the random variable $\bar{\theta}^*$ should be **closely approximated by a Gaussian**, and you can see from the qqplot above that this is true.

```
> truth <- alpha / ( alpha + beta )
> theta.bar.SE <- theta.out[ , 2 ]
> sum( ( theta.bar - 1.96 * theta.bar.SE < truth ) *
>   ( truth < theta.bar + 1.96 * theta.bar.SE ) ) / n.sim
> [1] 0.94
```

With this set of **pseudo-random numbers**, **94%** of the nominal **95% Monte Carlo confidence intervals** for the posterior mean **included the truth**.

Evidently **frequentist ideas** can be used to work out how big m needs to be to have **any desired** **Monte Carlo accuracy** for $\bar{\theta}^*$ as an estimate of the posterior mean $E(\theta|y)$.

In practice, with $p(\theta|y)$ unknown, I would probably take an **initial sample** (in this case, of size $m = 100$) and look at the MCSE to decide **how big** m **really needs to be**.

IID Example (continued)

Let's say I ran the program with `n.sim` $= 1$ and $m = 100$ and got the following **results**:

```
> theta.bar <- beta.sim( m, alpha, beta, 1, seed )
> theta.bar
          [,1]        [,2]
[1,] 0.1756400 0.001854220
```

(1) Suppose I wanted the MCSE of $\bar{\theta}^*$ to be (say) $\epsilon = 0.00005$; then I could **solve the equation**

$$\frac{\hat{\sigma}}{\sqrt{m}} = \epsilon \quad \leftrightarrow \quad m = \frac{\hat{\sigma}^2}{\epsilon^2}, \tag{3}$$

which says (unhappily) that the required m goes up as the **square** of the posterior SD and as the **inverse square** of ϵ.

The program results above show that $\frac{\hat{\sigma}}{\sqrt{100}} \doteq 0.001854220$, from which $\hat{\sigma} \doteq 0.01854220$, meaning that **to get** $\epsilon = 0.00005$ **I need a sample of size** $\frac{0.01854220^2}{0.00005^2} \doteq 137,525 \doteq 138\text{K}$.

IID Sample Size Determination

(2) Suppose instead that I wanted $\bar{\theta}^*$ to **differ** from the true posterior mean μ by **no more than** ϵ_1 with Monte Carlo probability **at least** $(1 - \epsilon_2)$:

$$P\left(\left| \bar{\theta}^* - \mu \right| \le \epsilon_1 \right) \ge 1 - \epsilon_2, \tag{4}$$

where $P(\cdot)$ here is based on the (frequentist) **Monte Carlo randomness** inherent in $\bar{\theta}^*$.

I know from the CLT and the calculations above that **in repeated sampling** $\bar{\theta}^*$ is approximately **Gaussian** with mean μ and variance $\frac{\sigma^2}{m}$; this leads to the inequality

$$m \ge \frac{\sigma^2 \left[\Phi^{-1} \left(1 - \frac{\epsilon_2}{2} \right) \right]^2}{\epsilon_1^2}, \tag{5}$$

where $\Phi^{-1}(q)$ is the place on the standard normal curve where $100q\%$ of the area is to the left of that place (the qth **quantile** of the standard Gaussian distribution).

(5) is like (3) except that the value of m from (3) has to be multiplied by $\left[\Phi^{-1} \left(1 - \frac{\epsilon_2}{2} \right) \right]^2$, which typically makes the required sample sizes **even bigger**.

A Closer Look at IID Sampling

For example, with $\epsilon_1 = 0.00005$ and $\epsilon_2 = 0.05$ — i.e., to have at least 95% Monte Carlo confidence that reporting the posterior mean as 0.1756 will be correct to about **four significant figures** — (5) says that I would need a monitoring run of at least $137,525(1.959964)^2 \doteq 528,296 \doteq 528\text{K}$.

This sounds like a long monitoring run but only takes about **2 seconds** at 1.6 Unix GHz, yielding $\left[\bar{\theta}^*, \widehat{SE}\left(\bar{\theta}^*\right)\right] = (0.1779052, 0.00002)$, which **compares favorably** with the **true value** 0.1779070.

It's evident from calculations like these that people often **report simulation-based answers** with numbers of significant figures **far in excess of what's justified** by the actual accuracy of the Monte Carlo estimates.

A closer look at IID sampling. I was able to easily perform the above **simulation study** because `R` has a large variety of built-in functions like `rbeta` for **pseudo-random-number generation**.

How would I go about **writing** such functions **myself**?

There are a number of **general-purpose** methods for generating random numbers (I won't attempt a survey here); the one we need to look closely at, to

Rejection Sampling

understand the algorithms that arise later in this part of the short course, is **rejection sampling** (von Neumann 1951), which is often one of the most **computationally efficient** ways to make IID draws from a distribution.

Example. Continuing the **AMI mortality case study** from Part 2, consider an **alternative prior specification** in which I'd like to put most (**90%**, say) of the prior mass in the interval **(0.05, 0.50)**; calculations like those in Part 2 within the **conjugate Beta family** yield **prior hyperparameter values** of $(\alpha_0, \beta_0) = \mathbf{(2.0,\ 6.4)}$ (this Beta distribution has prior mean and SD **0.24** and **0.14**, respectively).

Suppose that the sample size n was smaller at 74, and $s = 16$ AMI deaths were observed, so that the data mean was **0.216**; the posterior is then $\text{Beta}(\alpha_0 + s, \beta_0 + n - s) = \mathbf{Beta(18.0,\ 64.4)}$.

I'll pretend for the sake of illustration of **rejection sampling** that I don't know the formulas for the mean and SD of a Beta distribution, and suppose that I wanted to use **IID Monte Carlo sampling** from the $\text{Beta}(\alpha_0 + s, \beta_0 + n - s)$ posterior to estimate the **posterior mean**.

Rejection Sampling (continued)

Here's von Neumann's **basic idea**, which (as it turns out) works equally well for **scalar** or **vector** θ: suppose the target density $p(\theta|y)$ is **difficult** to sample from, but you can find an integrable **envelope function** $G(\theta|y)$ such that

(a) G **dominates** p in the sense that $G(\theta|y) \geq p(\theta|y) \geq 0$ for all θ and

(b) the density g obtained by normalizing G — later to be called the **proposal distribution** — is easy and fast to sample from.

Then to get a **random draw** from p, make a draw θ^* from g instead and **accept** or **reject** it according to an **acceptance probability** $\alpha_R(\theta^*|y)$; if you **reject** the draw, **repeat** this process until you accept.

von Neumann showed that the **choice**

$$\alpha_R(\theta^*|y) = \frac{p(\theta^*|y)}{G(\theta^*|y)} \tag{6}$$

correctly produces IID draws from p, and you can **intuitively** see that he's right by the following argument.

Making a **draw** from the posterior distribution of interest is like choosing a

Rejection Sampling (continued)

point **at random** (in two dimensions) under the density curve $p(\theta|y)$ in such a way that **all possible points are equally likely**, and then writing down its θ value.

If you instead draw from G so that all points under G are equally likely, to get **correct** draws from p you'll need to throw away any point that falls between p and G, and this can be accomplished by **accepting** each sampled point θ^* with probability $\frac{p(\theta^*|y)}{G(\theta^*|y)}$, as von Neumann said.

A **summary** of this method is on the next page.

The **figure** two pages below demonstrates this method on the Beta$(18.0, 64.4)$ density arising in the **Beta-Bernoulli example** above.

Rejection sampling permits considerable **flexibility** in the choice of **envelope function**; here, borrowing an idea from Gilks and Wild (1992), I've noted that the relevant Beta density is log concave (a real-valued function is log concave if its **second derivative** on the log scale is **everywhere non-positive**), meaning that it's easy to construct an envelope on that scale in a **piecewise linear** fashion, by choosing points on the log density and constructing

Rejection Sampling (continued)

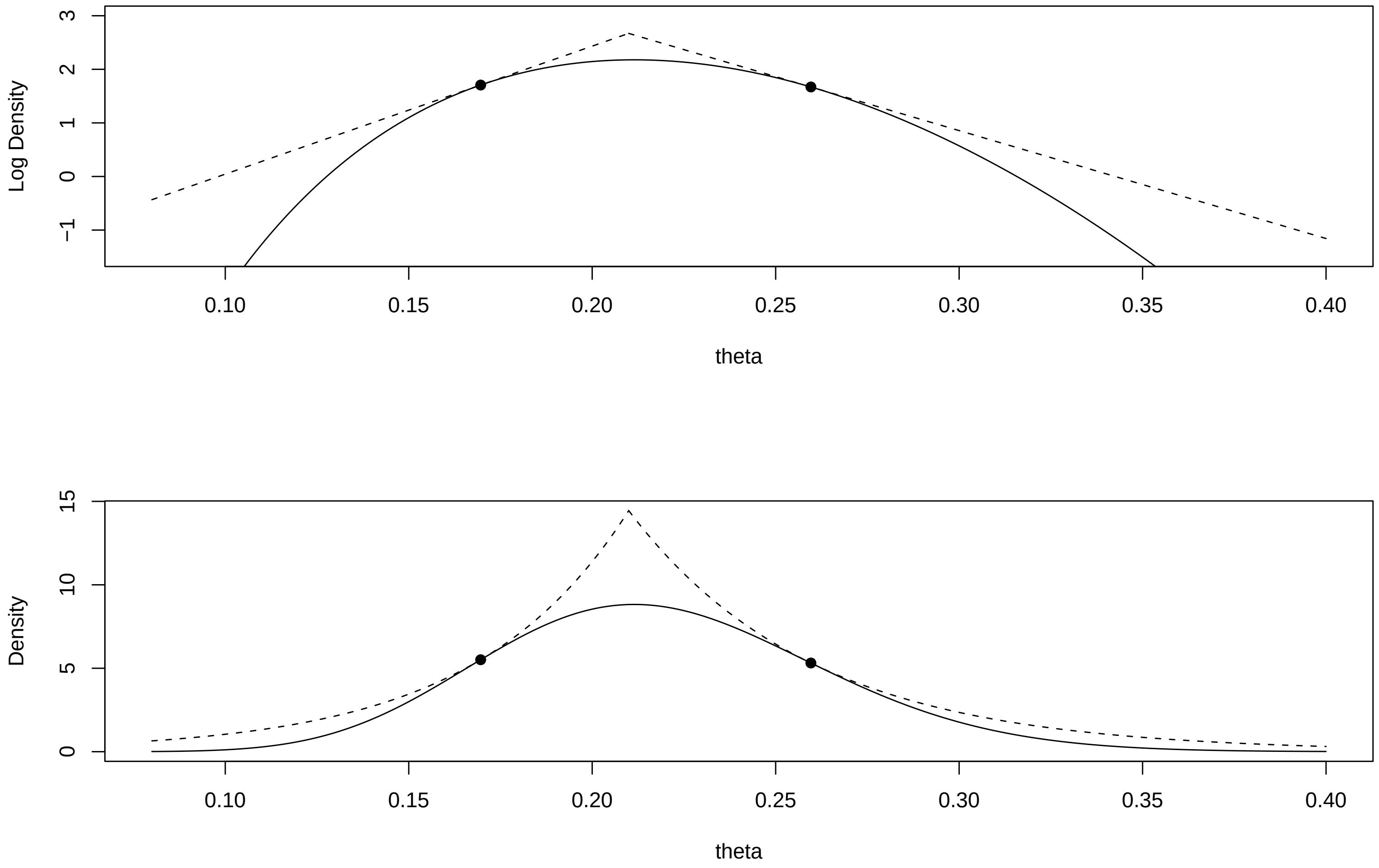
Algorithm (rejection sampling). To make m draws at random from the density $p(\theta|y)$ for scalar or vector θ, select an integrable **envelope function** G — which when normalized to integrate to 1 is the **proposal distribution** g — such that $G(\theta|y) \geq p(\theta|y) \geq 0$ for all θ; define the acceptance probability $\alpha_R(\theta^*|y) = \frac{p(\theta^*|y)}{G(\theta^*|y)}$; and

```
    Initialize $t \leftarrow 0$
    Repeat {
        Sample $\theta^* \sim g(\theta|y)$
        Sample $u \sim$ Uniform$(0, 1)$
        If $u \leq \alpha_R(\theta^*|y)$ then
            { $\theta_{t+1} \leftarrow \theta^*$;  $t \leftarrow (t+1)$ }
    }
    until $t = m$.
```

tangents to the curve at those points.

The **simplest** possible such envelope involves **two line segments**, one on either side of the **mode**.

Rejection Sampling (continued)



The optimal choice of the tangent points would maximize the marginal **probability of acceptance** of a draw in the rejection algorithm, which can be shown to be

Rejection Sampling (continued)

$$\left[\int G(\theta)\, d\theta \right]^{-1} ; \qquad (7)$$

in other words, you should **minimize** the area under the (un-normalized) envelope function subject to the constraint that it **dominates** the target density $p(\theta|y)$ (which makes eminently good sense).

Here this optimum turns out to be attained by locating the two tangent points at about **0.17** and **0.26**, as in the figure above; the resulting acceptance probability of about **0.75** could clearly be **improved** by adding more tangents.

Piecewise linear envelope functions on the log scale are a **good choice** because the resulting envelope density on the raw scale is a piecewise set of **scaled exponential distributions** (see the bottom panel in the figure above), from which random samples can be taken easily and **quickly**.

A **preliminary** sample of $m_0 = 500$ IID draws from the Beta$(18.0, 64.4)$ distribution using the above rejection sampling method yields $\bar{\theta}^* = \mathbf{0.2197}$ and $\hat{\sigma} = \mathbf{0.04505}$, meaning that the posterior mean has already been estimated with an **MCSE** of only $\frac{\hat{\sigma}}{\sqrt{m_0}} = 0.002$ even with just **500** draws.

Rejection Sampling (continued)

Suppose, however, that — as in equation (4) above — I want $\bar{\theta}^*$ to **differ** from the true posterior mean μ by no more than some (perhaps even smaller) **tolerance** ϵ_1 with Monte Carlo probability at least $(1 - \epsilon_2)$; then equation (5) tells me how long to **monitor** the simulation output.

For instance, to pin down **three significant figures** (sigfigs) in the posterior mean in this example with high Monte Carlo accuracy I might take $\epsilon_1 = 0.0005$ and $\epsilon_2 = 0.05$, which yields a **recommended IID sample size** of $\frac{(0.04505^2)(1.96)^2}{0.0005^2} \doteq 31,200$.

So I take another sample of **30,700** (which is virtually instantaneous at 1.6 Unix GHz) and **merge** it with the 500 draws I already have; this yields $\bar{\theta}^* = 0.21827$ and $\hat{\sigma} = 0.04528$, meaning that the **MCSE** of this estimate of μ is $\frac{0.04528}{\sqrt{31200}} \doteq 0.00026$.

I might **announce** that I think $E(\theta|y)$ is about **0.2183**, give or take about **0.0003**, which accords well with the true value **0.2184**.

Of course, **other aspects** of $p(\theta|y)$ are equally easy to monitor; for example, if I want a Monte Carlo estimate of $p(\theta \leq q|y)$ for some q, as noted above I just work out the **proportion** of the sampled θ^* values that are no larger than q.

Beyond Rejection Sampling

Or, even better, I recall that $P(A) = E[I(A)]$ for any event or proposition A, so to the **Monte Carlo dataset** (see page 35 below) consisting of 31,200 rows and one column (the θ_t^*) I add a column monitoring the values of the **derived variable** which is 1 whenever $\theta_t^* \leq q$ and 0 otherwise; the **mean** of this derived variable is the Monte Carlo estimate of $p(\theta \leq q|y)$, and I can attach an **MCSE** to it in the same way I did with $\bar{\theta}^*$.

By this approach, for instance, the **Monte Carlo estimate** of $p(\theta \leq 0.15|y)$ based on the 31,200 draws examined above comes out $\hat{p} = \mathbf{0.0556}$ with an MCSE of **0.0013**.

Percentiles are typically harder to pin down with equal Monte Carlo accuracy (in terms of sigfigs) than means or SDs, because the 0/1 scale on which they're based is **less information-rich** than the θ^* scale itself; if I wanted an MCSE for $\hat{p}$ of 0.0001 I would need an IID sample of more than **5 million draws** (which would still only take a **few seconds** at contemporary workstation speeds).

IID sampling is not necessary. Nothing in the Metropolis-Ulam idea of

MCMC

Monte Carlo estimates of posterior summaries requires that these estimates be based on **IID samples from the posterior**.

This is lucky, because in practice it's often difficult, particularly when θ is a **vector of high dimension** (say k), to figure out how to make such an IID sample, via rejection sampling or other methods (e.g., imagine trying to find an **envelope function** for $p(\theta|y)$ when k is 10 or 100 or **1,000**).

Thus it's necessary to **relax** the assumption that $\theta_j^* \overset{\text{IID}}{\sim} p(\theta|y)$, and to consider samples $\theta_1^*, \ldots, \theta_m^*$ that form a **time series**: a series of draws from $p(\theta|y)$ in which θ_j^* may **depend on** $\theta_{j'}^*$ for $j' < j$.

In their pioneering paper Metropolis et al. (1953) allowed for **serial dependence** of the θ_j^* by combining von Neumann's idea of rejection sampling (which had itself only been published a few years earlier in 1951) with concepts from **Markov chains**, a subject in the theory of **stochastic processes**.

Combining **Monte Carlo sampling** with **Markov chains** gives rise to the name now used for this technique for solving the Bayesian high-dimensional integration problem: **Markov chain Monte Carlo** (MCMC).

3.3 Brief Review of Markov Chains

Markov chains. A **stochastic process** is just a collection of random variables $\{\theta_t^*, t \in T\}$ for some **index set** T, usually meant to stand for **time**.

In practice T can be either **discrete**, e.g., $\{0, 1, \ldots\}$, or **continuous**, e.g., $[0, \infty)$.

Markov chains are a special kind of stochastic process that can either unfold in discrete or continuous time — I'll talk here about **discrete-time Markov chains**, which is all you need for MCMC.

The **possible values** that a stochastic process can take on are collectively called the **state space** S of the process — in the simplest case S is **real-valued** and can also either be discrete or continuous.

Intuitively speaking, a Markov chain (e.g., Feller, 1968; Roberts, 1996; Gamerman, 1997) is a stochastic process evolving in time in such a way that the **past and future states of the process are independent given the present state**—in other words, to figure out where the chain is likely to go next you don't need to pay attention to where it's been, you just need to consider **where it is now**.

Markov Chains (continued)

More formally, a stochastic process $\{\theta_t^*, t \in T\}$, $T = \{0, 1, \dots\}$, with state space S is a **Markov chain** if, for any set $A \in S$,

$$P(\theta_{t+1}^* \in A | \theta_0^*, \dots, \theta_t^*) = P(\theta_{t+1}^* \in A | \theta_t^*). \tag{8}$$

The theory of Markov chains is **harder mathematically** if S is continuous (e.g., Tierney, 1996), which is what we need for MCMC with real-valued parameters, but **most of the main ideas emerge with discrete state spaces**, and I'll assume discrete S in the intuitive discussion here.

Example. For a simple example of a **discrete-time Markov chain** with a **discrete state space**, imagine a **particle** that moves around on the integers $\{\dots, -2, -1, 0, 1, 2, \dots\}$, starting at 0 (say).

Wherever it finds itself at time t—say at i—it **tosses a (3-sided) coin** and moves to $(i-1)$ with probability p_1, stays at i with probability p_2, and moves to $(i+1)$ with probability p_3, for some $0 < p_1, p_2, p_3 < 1$ with $p_1 + p_2 + p_3 = 1$—these are the **transition probabilities** for the process.

This is a **random walk** (on the integers), and it's **clearly a Markov chain**.

Markov Chains (continued)

Nice behavior. The most **nicely-behaved** Markov chains satisfy **three properties**:

- They're **irreducible**, which basically means that no matter where it starts the chain has to be able to reach any other state in a finite number of iterations with positive probability;
- They're **aperiodic**, meaning that for all states i the set of possible **sojourn times**, to get back to i having just left it, can have no divisor bigger than 1 (this is a **technical** condition; periodic chains still have some nice properties, but the nicest chains are aperiodic).
- They're **positive recurrent**, meaning that (a) for all states i, if the process starts at i it will return to i with probability 1, and (b) the expected length of waiting time til the first return to i is finite.

Notice that this is a bit delicate: wherever the chain is now, we insist that it **must certainly come back here**, but we don't expect to have to **wait forever** for this to happen.

Markov Chains (continued)

The random walk defined above is clearly **irreducible** and **aperiodic**, but it may not be **positive recurrent** (depending on the p_i): it's true that it has positive probability of returning to wherever it started, but (because S is **unbounded**) this probability may not be 1, and on average you may have to wait forever for it to return.

We can fix this by **bounding** S: suppose instead that $S = \{-k, -(k-1), \ldots, -1, 0, 1, \ldots, k\}$, keeping the same transition probabilities except **rejecting** any moves **outside the boundaries** of S.

This bounded random walk now satisfies **all three of the nice properties**.

The value of nice behavior. Imagine running the bounded random walk for a long time, and look at the **distribution** of the **states** it visits—over time this distribution should **settle down** (converge) to a kind of limiting, **steady-state** behavior.

This can be demonstrated by **simulation**, for instance in `R`, and using the **bounded random walk** as an example:

Markov Chains (continued)

```
rw.sim <- function( k, p, theta.start, n.sim, seed ) {
  set.seed( seed )
  theta <- rep( 0, n.sim + 1 )
  theta[ 1 ] <- theta.start
  for ( i in 1:n.sim ) {
    theta[ i + 1 ] <- move( k, p, theta[ i ] )
  }
  return( table( theta ) )
}
move <- function( k, p, theta ) {
  repeat {
    increment <- sample( x = c( -1, 0, 1 ), size = 1, prob = p )
    theta.next <- theta + increment
    if ( abs( theta.next ) <= k ) {
      return( theta.next )
      break
    }
  }
}
```

Markov Chains (continued)

```
greco 171> R
R version 2.5.1 (2007-06-27)
Copyright (C) 2007 The R Foundation for Statistical Computing
> p <- c( 1, 1, 1 ) / 3
> k <- 5
> theta.start <- 0
> seed <- c( 6425451, 9626954 )
> rw.sim( k, p, theta.start, 10, seed )
theta
0 1 2
5 5 1
> rw.sim( k, p, theta.start, 100, seed )
-2 -1  0  1  2  3  4  5
 7  9 16 17 23 14  8  7
> rw.sim( k, p, theta.start, 1000, seed )
 -5  -4  -3  -2  -1   0   1   2   3   4   5
 65 115 123 157 148 123 106  82  46  21  15
> rw.sim( k, p, theta.start, 10000, seed )
  -5   -4   -3   -2   -1    0    1    2    3    4    5
 581  877  941  976  959 1034 1009  982 1002  959  681
```

Markov Chains (continued)

```
> rw.sim( k, p, theta.start, 100000, seed )
  -5   -4   -3   -2   -1    0    1    2    3    4    5
6515 9879 9876 9631 9376 9712 9965 9749 9672 9352 6274
> rw.sim( k, p, theta.start, 1000000, seed )
   -5    -4    -3    -2    -1     0     1     2     3     4     5
65273 98535 97715 96708 95777 96607 96719 96361 96836 95703 63767
```

You can see that the distribution of where the chain has visited is **converging** to something close to **uniform** on $\{-5, -4, \ldots, 4, 5\}$, except for the effects of the **boundaries**.

Letting q_1 denote the **limiting** probability of being in one of the 9 **non-boundary** states $(-4, -3, \ldots, 3, 4)$ and q_2 be the **long-run** probability of being in one of the 2 **boundary** states $(-5, 5)$, on grounds of **symmetry** you can guess that q_1 and q_2 should satisfy

$$9 q_1 + 2 q_2 = 1 \quad \text{and} \quad q_1 = \frac{3}{2} q_2, \tag{9}$$

from which $(q_1, q_2) = \left(\frac{3}{31}, \frac{2}{31} \right) \doteq (0.096774, 0.064516)$.

Based on the run of **1,000,001 iterations** above you would estimate these

Markov Chains (continued)

probabilities **empirically** as

$$\left[\frac{98535+\ldots+95703}{(9)(1000001)}, \frac{65273+63767}{(2)(1000001)} \right] \doteq (0.096773, 0.064520).$$

It should also be clear that the limiting distribution **does not depend** on the initial value of the chain:

```
> rw.sim( k, p, 5, 100000, seed )
  -5   -4   -3   -2   -1    0    1    2    3    4    5
6515 9879 9876 9624 9374 9705 9959 9738 9678 9365 6288
```

Of course, you get a **different limiting distribution** with a **different choice of** (p_1, p_2, p_3):

```
> p <- c( 0.2, 0.3, 0.5 )
> rw.sim( k, p, 0, 10, seed )
0 1 2 3
1 3 4 3
> rw.sim( k, p, 0, 100, seed )
 0  1  2  3  4  5
 1  3  6 13 30 48
```

Markov Chains (continued)

```
> rw.sim( k, p, 0, 1000, seed )
  0   1   2   3   4   5
  1  18  71 157 336 418
> rw.sim( k, p, 0, 10000, seed )
  -5   -4   -3   -2   -1    0    1    2    3    4    5
   5   16   19   30   28   74  215  583 1344 3470 4217
> rw.sim( k, p, 0, 100000, seed )
   -5    -4    -3    -2    -1     0     1     2     3     4     5
    5    22    53   132   302   834  2204  5502 13489 34460 42998
> rw.sim( k, p, 0, 1000000, seed )
-5     -4     -3     -2     -1      0      1      2      3      4      5
61    198    511   1380   3398   8591  22117  54872 137209 343228 428436
```

Stationary distributions. A positive recurrent and aperiodic chain is called **ergodic**, and it turns out that such chains possess a unique **stationary** (or **equilibrium**, or **invariant**) distribution π, characterized by the relation

$$\pi(j) = \sum_i \pi(i) P_{ij}(t) \qquad (10)$$

for all states j and times $t \geq 0$, where $P_{ij}(t) = P(\theta_t^* = j | \theta_{t-1}^* = i)$ is the **transition matrix** of the chain.

The MCMC Payoff

Informally, the stationary distribution characterizes the **behavior that the chain will settle into** after it's been run for a long time, regardless of its initial state.

The point of all of this. Given a parameter vector θ and a data vector y, the Metropolis et al. (1953) idea is to **simulate** random draws from the posterior distribution $p(\theta|y)$, by constructing a **Markov chain** with the following four properties:

- It should have the **same state space** as θ,
- It should be **easy to simulate from**,
- It should work **equally well** with an **un-normalized** $p(\theta|y)$, so that it's **not necessary to evaluate the normalizing constant**, and
- Its **equilibrium distribution** should be $p(\theta|y)$.

If you can do this, you can run the Markov chain for a long time, generating a huge sample from the posterior, and then use **simple descriptive summaries** (means, SDs, correlations, histograms or kernel density estimates) to extract any features of the posterior you want.

The Ergodic Theorem

The mathematical fact that underpins this strategy is the ergodic theorem: if the Markov chain $\{\theta_t^*\}$ is ergodic and f is any real-valued function for which $E_\pi|f(\theta)|$ is finite, then with probability 1 as $m \rightarrow \infty$

$$\frac{1}{m}\sum_{t=1}^{m} f(\theta_t^*) \rightarrow E_\pi[f(\theta)] = \sum_i f(i)\,\pi(i), \tag{11}$$

in which the right side is just the **expectation** of $f(\theta)$ under the stationary distribution π.

In plain English this means that — as long as the stationary distribution **is** $p(\theta|y)$ — you can learn (to arbitrary accuracy) about things like posterior means, SDs, and so on just by **waiting for stationarity to kick in** and **monitoring** thereafter **for a long enough period**.

Of course, as Roberts (1996) notes, the theorem is **silent** on the two key practical questions it raises: **how long you have to wait** for stationarity, and **how long to monitor** after that.

A third practical issue is what to use for the **initial value** θ_0^*: intuitively the

The Monte Carlo and MCMC Datasets

closer θ_0^* is to the **center** of $p(\theta|y)$ the **less time** you should have to wait for stationarity.

The standard way to deal with **waiting for stationarity** is to (a) run the chain from a **good starting value** θ_0^* for b iterations, until **equilibrium** has been reached, and (b) **discard** this initial **burn-in** period.

All of this motivates the topic of **MCMC diagnostics**, which are intended to answer the following questions:

- What should I use for the **initial value** θ_0^*?
- How do I know when I've reached **equilibrium**? (This is equivalent to asking **how big** b should be.)
- Once I've reached equilibrium, how big should m be, i.e., how long should I **monitor the chain** to get posterior summaries with **decent accuracy**?

The Monte Carlo and MCMC datasets. The basis of the Monte Carlo approach to obtaining **numerical approximations** to posterior summaries like means and SDs is the (weak) **Law of Large Numbers**: with IID sampling

The Monte Carlo and MCMC Datasets (continued)

the **Monte Carlo estimates** of the true summaries of $p(\theta|y)$ are **consistent**, meaning that they can be made arbitrarily close to the truth with arbitrarily high probability as the number of monitoring iterations $m \rightarrow \infty$.

Before we look at how Metropolis et al. attempted to achieve the same goal with a **non-IID Monte Carlo approach**, let's look at the **practical consequences** of switching from IID to Markovian sampling.

Running the **IID rejection sampler** on the AMI mortality example above for a total of m monitoring iterations would produce something that might be called the **Monte Carlo (MC) dataset**, with one **row** for each **iteration** and one **column** for each **monitored quantity**; in that example it might look like the table on the next page (MCSEs in parenthesis).

Running the **Metropolis sampler** on the same example would produce something that might be called the **MCMC dataset**.

It would have a **similar structure** as far as the **columns** are concerned, but the rows would be divided into **three phases**:

- Iteration 0 would be the value(s) used to **initialize** the Markov chain;

The MC and MCMC Data Sets

The MC Data Set:

Iteration	θ	$I(\theta \leq 0.15)$
1	$\theta_1^* = 0.244$	$I_1^* = 0$
2	$\theta_2^* = 0.137$	$I_2^* = 1$
$\vdots$	$\vdots$	$\vdots$
$m = 31,200$	$\theta_m^* = 0.320$	$I_m^* = 0$
Mean	0.2183 (0.003)	0.0556 (0.0013)
SD	0.04528	—
Density Trace	(like the bottom plot on page 17)	—

- Iterations 1 through b would be the **burn-in** period, during which the chain reaches its **equilibrium** or **stationary** distribution (as mentioned above, iterations 0 through b are generally **discarded**); and

- Iterations $(b + 1)$ through $(b + m)$ would be the **monitoring** run, on which **summaries** of the posterior (means, SDs, density traces, ...) will be based.

A Metropolis Example ($k = 1$)

Here's an **example** that can serve as the basis of a **fairly general strategy** for **Metropolis sampling** when the **unknown** θ is a **vector of real numbers** of length $k \ge 1$; I'll first give **details** in a situation with $k = 1$, and then we'll look at how to **generalize** this to $k > 1$.

$\boxed{\text{Example:}}$ The setup is **one-sample Gaussian data** with **known** μ, **unknown** $\theta = \sigma^2$, and **little information** about θ **external** to the **data set** $y = (y_1, \dots, y_n)$:

$$\begin{aligned} \sigma^2 &\sim \text{diffuse} \\ (y_i | \sigma^2) &\overset{\text{IID}}{\sim} N\left(\mu, \sigma^2\right) \quad (i = 1, \dots, n)\,. \end{aligned} \tag{12}$$

The **joint sampling distribution** is

$$\begin{aligned} p(y|\sigma^2) &= \prod_{i=1}^{n} \frac{1}{\sigma\sqrt{2\pi}} \exp\left[-\frac{1}{2\,\sigma^2}(y_i - \mu)^2\right] \\ &= c\left(\sigma^2\right)^{-\frac{n}{2}} \exp\left[-\frac{1}{2\,\sigma^2}\sum_{i=1}^{n}(y_i - \mu)^2\right]. \end{aligned} \tag{13}$$

Metropolis = Symmetric Proposal Distribution

In **homework 3** you showed that the **MLE** for σ^2 in this **model** is

$$\hat{\sigma}^2 = \frac{1}{n} \sum_{i=1}^{n} (y_i - \mu)^2 \, , \tag{14}$$

so that the **joint sampling distribution** can also be written as

$$p(y|\sigma^2) = c \left(\sigma^2 \right)^{-\frac{n}{2}} \exp \left[-\frac{n \, \hat{\sigma}^2}{2 \, \sigma^2} \right] ; \tag{15}$$

in the **alternative parameterization** $\theta = \sigma^2$ this is

$$p(y|\theta) = c \, \theta^{-\frac{n}{2}} \exp \left[-\frac{n \, \hat{\sigma}^2}{2 \, \theta} \right] . \tag{16}$$

Now **Metropolis sampling** is based on a **symmetric proposal distribution**, and θ lives on the **positive part** of the real line; if you use a **symmetric** proposal distribution on θ, you'll have to **reject any proposed moves below 0**, and this can be **Monte-Carlo inefficient**.

An easy way to **solve** this problem is to **propose moves** not on θ but on a **transformed version** of θ that **lives on the whole real line**; the **obvious candidate** is $\lambda = \log \theta$, so that $\theta = e^\lambda$.

Need the Prior on the Transformed Parameter

In this **parameterization** the **sampling distribution** (16) is

$$p(y|\lambda) = c\, e^{-\frac{n\lambda}{2}} \exp\left[-\frac{n\,\hat{\sigma}^2}{2\,e^{\lambda}} \right] \tag{17}$$

(just **substitute** e^{λ} everywhere you see θ), so that the **likelihood function** in terms of λ is also

$$l(\lambda|y) = c\, e^{-\frac{n\lambda}{2}} \exp\left[-\frac{n\,\hat{\sigma}^2}{2\,e^{\lambda}} \right] \, ; \tag{18}$$

thus the **log likelihood** in this parameterization is

$$ll(\lambda|y) = c - \frac{n\,\lambda}{2} - \frac{n\,\hat{\sigma}^2}{2\,e^{\lambda}} \, . \tag{19}$$

From **Bayes's Theorem**,

$$\log p(\lambda|y) = c + \log p(\lambda) + ll(\lambda|y) \, , \tag{20}$$

so to define the **log posterior** analytically we need to work out the **prior** for λ that's **implied** (from the **change of variables formula**) by a **suitably-chosen diffuse prior** for θ.

Need the Change-of-Variables Formula For the Prior

In homework 3 I mentioned that a **popular diffuse prior** for $\theta = \sigma^2$ in this setup is the **improper prior**

$$p(\theta) = c\,\theta^{-1}\,; \tag{21}$$

let's see what this **corresponds to** as a prior on $\lambda = \log\theta$.

A **quick way** to use the change-of-variables formula is through the **expression**

$$p(\theta)\,|d\theta| = p(\lambda)\,|d\lambda|\,, \quad \text{from which} \quad p(\lambda) = p(\theta)\left|\frac{d\theta}{d\lambda}\right|. \tag{22}$$

Now **in this case** $\theta = e^\lambda$, so that $\left|\frac{d\theta}{d\lambda}\right| = e^\lambda$, and (in terms of λ) $p(\theta) = c\,\theta^{-1} = c\,\left(e^\lambda\right)^{-1} = c\,e^{-\lambda}$, so that

$$p(\lambda) = c\,\theta^{-1}\,e^\lambda = c\,e^{-\lambda}\,e^\lambda = c\,. \tag{23}$$

Thus we see why this is a **popular diffuse prior** for $\theta = \sigma^2$: it corresponds to the **(improper) uniform prior** on $(-\infty, \infty)$ for $\lambda = \log\theta$ (which is a **natural choice for diffuseness** on the $\log\theta$ scale).

Random-Walk Metropolis, With a Tuning Constant

Now I'm ready to think about what to use for my **symmetric proposal distribution** $g(\theta^* | \theta_t, y)$. which tells me how to **sample** a value θ^* as a **possible place** for the **Markov chain** to go **next**, given that it's **currently** at θ_t; **symmetry** here means that $g(\theta^* | \theta_t, y) = g(\theta_t | \theta^*, y)$.

A choice that's simultaneously **simple** and **reasonably Monte-Carlo efficient** is what people call random-walk Metropolis: I'll **center** the **proposal distribution** at **where I am now** (θ_t), and then make a **draw** from a distribution that's **symmetric** about that point.

Metropolis et al. (1953) used the **uniform** distribution $U(\theta_t - c, \theta_t + c)$; these days people tend to use a **Gaussian** distribution $N\left(\theta_t, \sigma_*^2\right)$; in both cases c and σ_* are **tuning constants**, which we can choose to **optimize** the **Monte Carlo efficiency** of the resulting **Metropolis sampler**.

Let's call σ_* the **proposal distribution standard deviation** (PDSD); with that convention I'm ready to write `R` code to **implement the algorithm**.

I'll need a **driver function** that does the **sampling** and generates the **MCMC data set**; this will depend on a **function** that calculates

R Implementation

the **acceptance probabilities**; this will in turn depend on a **function** to calculate the **log posterior**; and this will finally depend on functions to calculate the log prior and log likelihood.

```
metropolis.example <- function( y, mu, sigma.star, sigma2.0, M, B ) {
  n <- length( y )
  sigma2.hat <- sum( ( y - mu )^2 ) / n
  lambda.hat <- log( sigma2.hat )
  mcmc.data.set <- matrix( NA, M + B + 1, 2 )
  mcmc.data.set[ 1, ] <- c( log( sigma2.0 ), sigma2.0 )
  acceptance.count <- 0
  for ( i in 2:( M + B + 1 ) ) {
    lambda.current <- mcmc.data.set[ i - 1, 1 ]
    lambda.star <- rnorm( 1, lambda.current, sigma.star )
    u <- runif( 1 )
    if ( u <= acceptance.probability( lambda.star, lambda.current,
      lambda.hat, n ) ) {
        mcmc.data.set[ i, 1 ] <- lambda.star
        mcmc.data.set[ i, 2 ] <- exp( lambda.star )
        acceptance.count <- acceptance.count + 1
    }
```

R Implementation (continued)

```
    else {
      mcmc.data.set[ i, 1 ] <- lambda.current
      mcmc.data.set[ i, 2 ] <- exp( lambda.current )
    }
    if ( ( i %% 1000 ) == 0 ) print( i )
  }
  print( acceptance.rate <- acceptance.count / ( M + B ) )
  return( mcmc.data.set )
}

acceptance.probability <- function( lambda.star, lambda.current,
  lambda.hat, n ) {
  return( exp( log.posterior( lambda.star, lambda.hat, n ) -
    log.posterior( lambda.current, lambda.hat, n ) ) )
}

log.posterior <- function( lambda, lambda.hat, n ) {
  return( log.prior( lambda ) + log.likelihood( lambda, lambda.hat,
    n ) )
}
```

NB10 Example

```
log.likelihood <- function( lambda, lambda.hat, n ) {
  return( - lambda * n / 2 - n * exp( lambda.hat ) /
    ( 2 * exp( lambda ) ) )
}

log.prior <- function( lambda ) {
  return( 0 )
}
```

Let's run this code on the **NB10 data**, (for illustration) taking μ to be the **sample mean** $\bar{y}$:

```
y <- c( 409., 400., 406., 399., 402., 406., 401., 403., 401., 403., 398.,
  403., 407., 402., 401., 399., 400., 401., 405., 402., 408., 399., 399.,
  402., 399., 397., 407., 401., 399., 401., 403., 400., 410., 401., 407.,
  423., 406., 406., 402., 405., 405., 409., 399., 402., 407., 406., 413.,
  409., 404., 402., 404., 406., 407., 405., 411., 410., 410., 410., 401.,
  402., 404., 405., 392., 407., 406., 404., 403., 408., 404., 407., 412.,
  406., 409., 400., 408., 404., 401., 404., 408., 406., 408., 406., 401.,
  412., 393., 437., 418., 415., 404., 401., 401., 407., 412., 375., 409.,
  406., 398., 406., 403., 404. )
```

Choosing the PDSD

```
print( mu <- mean( y ) )
[1] 404.59
print( sigma2.0 <- var( y ) )
[1] 41.8201
M <- 100000
B <- 500
```

Now I need to choose a **PDSD** σ_*: if I know something about the **scale** of the **posterior** I'm sampling from — e.g., if I've computed $\widehat{SE}\left(\hat{\theta}_{MLE}\right)$ — I can use this to set the **initial PDSD** intelligently (it's been shown that with $k = 1$ the **optimal PDSD** when the posterior is **approximately Gaussian** is $\sigma_* = 2.4\,\sigma_\theta$, where σ_θ is the **posterior SD** for θ), but if I don't know about the **scale** of $p(\theta|y)$ I can use **trial and error** and **tune the sampler** to have a **good acceptance rate** (it's also been shown that with $k = 1$ the **optimal acceptance rate** is about **44%**).

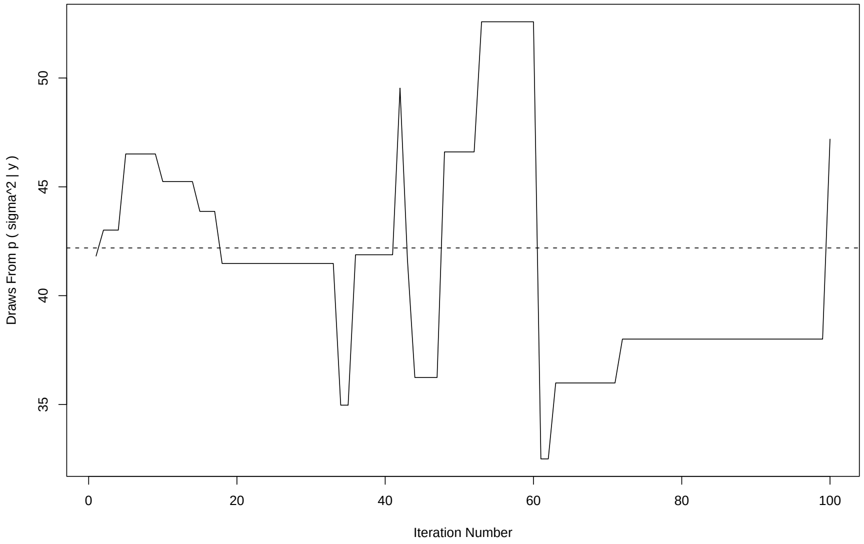
I'll **start out** with $\sigma_* = 1.0$ and see what the **acceptance rate** is:

```
sigma.star <- 1
mcmc.data.set.1 <- metropolis.example( y, mu, sigma.star, sigma2.0, M, B )
[1] 0.1747861
```

Blocky Output: PDSD Too Big (Acceptance Rate Too Low)

This took about **26 sec** at **1.6 Unix GHz** and produced the following **posterior summaries**:

```
mean( mcmc.data.set.1[ , 2 ] )
[1] 42.19285
sd( mcmc.data.set.1[ , 2 ] )
[1] 6.038515
plot( 1:100, mcmc.data.set.1[ 1:100, 2 ], type = 'l',
  xlab = 'Iteration Number', ylab = 'Draws From p ( sigma^2 | y )' )
```



This output is **blocky**: the **PDSD** is **too big** and the **acceptance rate** is **too low**.

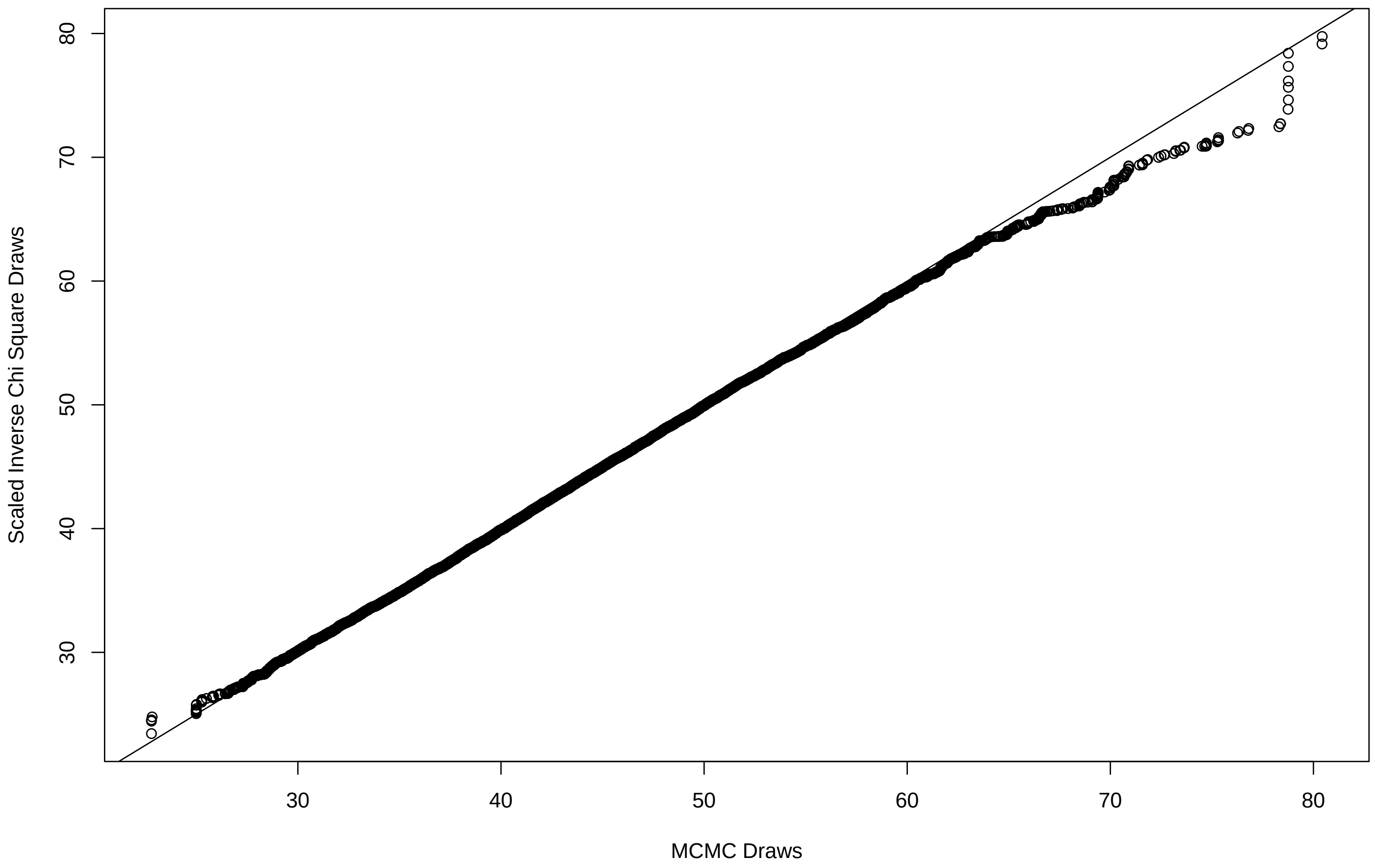
Demonstrating the Validity of the Sampler

In this problem we know the **right answer** for the **posterior** — $(\sigma^2|y) \sim \chi^{-2}\left(n, \hat{\sigma}^2\right)$ — so we can use this to **demonstrate** the **validity** of the **Metropolis sampler**:

```
rsichi2 <- function( n, nu, s2 ) {
  return( nu * s2 / rchisq( n, nu ) )
}
n <- 100
print( sigma2.hat <- ( n - 1 ) * var( y ) / n )
[1] 41.4019
direct.simulation <- rsichi2( 100000, 100, 41.4019 )
n * sigma2.hat / ( n - 2 )
[1]42.24684
qqplot( mcmc.data.set.1[ , 2 ], backup, xlab = ’MCMC Draws’,
  ylab = ’Scaled Inverse Chi Square Draws’ )
abline( 0, 1 )
```

The **MCMC posterior mean** is **correct** up to **Monte-Carlo noise** in the **third significant figure**, and the **qqplot** shows that we're **sampling** from the **right distribution**:

Demonstrating the Validity of the Sampler (continued)



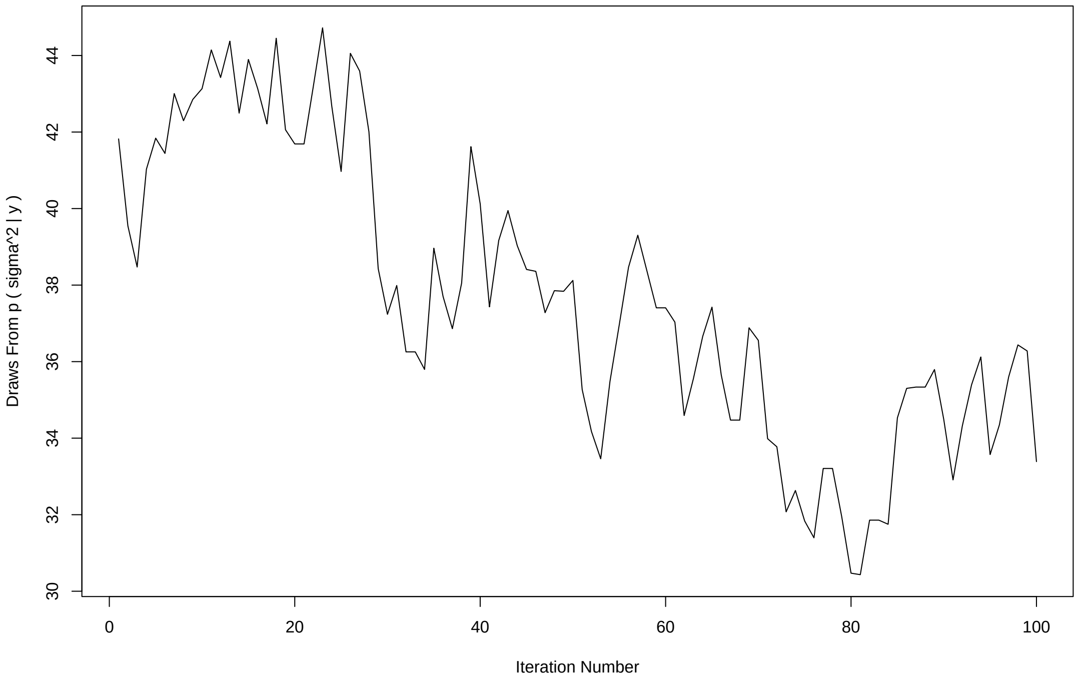
(The **discrepancy** in the **right tail** is just **Monte-Carlo noise**, as you can **verify** by (a) **sampling twice** with the `rchisq` function, (b) making a **qqplot** of the resulting two samples, and (c) **repeating** (a) and (b) a few times.)

Sticky Output: PDSD Too Small (Acceptance Rate Too Big)

For illustration, here's what happens with a **Metropolis sampler** when the **PDSD** is **too small**:

```
sigma.star <- 0.05
mcmc.data.set.2 <- metropolis.example( y, mu, sigma.star, sigma2.0, M, B )
[1] 0.8869552
plot( 1:100, mcmc.data.set.2[ 1:100, 2 ], type = 'l',
  xlab = 'Iteration Number', ylab = 'Draws From p ( sigma^2 | y )' )
```

Now the **acceptance rate** is **too high**, and the **sampler output** is sticky:



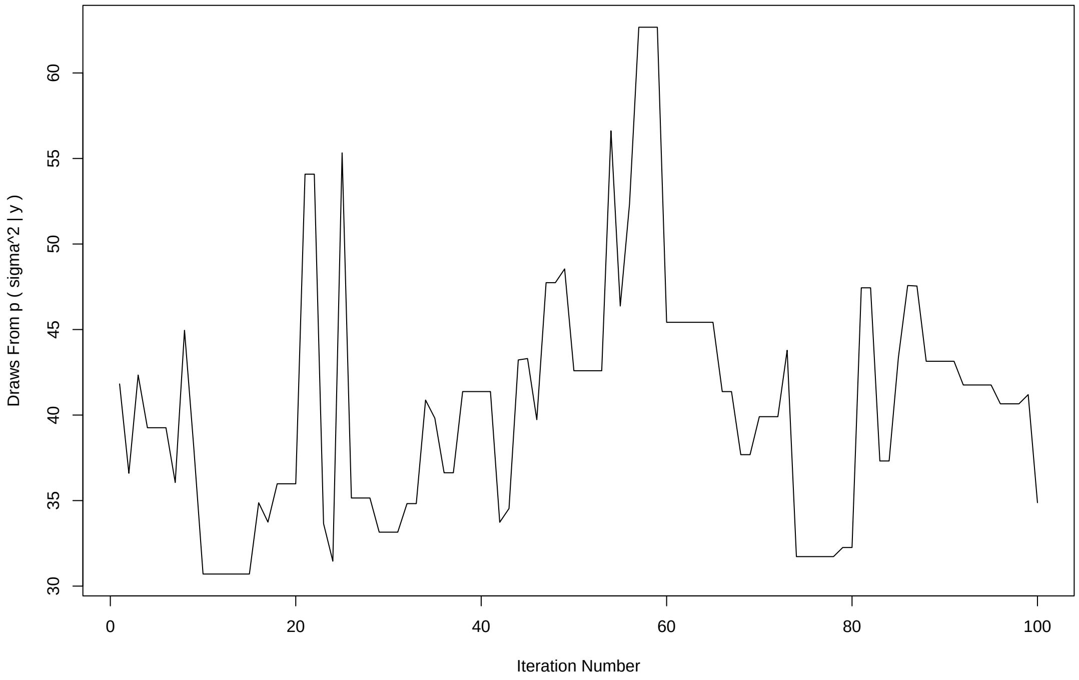
Optimized Random-Walk Metropolis Sampler

We can use the **result** mentioned above to optimize this process: using a **random-walk Metropolis sampler** with a **Gaussian proposal distribution** and an **approximately Gaussian target distribution**, to get the **optimal acceptance rate** of about **44%** the **PDSD** (on the **scale** on which the **moves** are made, which here is the $\log(\theta)$ scale) should be about **2.4** times bigger than the **posterior SD**:

```
2.4 * sd( mcmc.data.set.1[ , 1 ] )
[1] 0.3415815
sigma.star <- 0.34
mcmc.data.set.3 <- metropolis.example( y, mu, sigma.star, sigma2.0, M, B )
[1] 0.4420597
plot( 1:100, mcmc.data.set.3[ 1:100, 2 ], type = ’l’,
  xlab = ’Iteration Number’, ylab = ’Draws From p ( sigma^2 | y )’ )
```

The resulting **time series** (on the next page) is the **best we can do** in this **class** of **Metropolis samplers** for this problem; as **MCMC output** goes, this plot actually shows **pretty good mixing** of the **Markov chain** (there are relatively few **flat spots**, and the output is **not as sticky** as it was with $\sigma_* = 0.05$).

The MCMC 4–Plot



A **useful graphical diagnostic** for **each monitored quantity** θ_j in $\theta = (\theta_1, \ldots, \theta_k)$ is a **picture** that might be called an **MCMC 4–plot**, containing

(a) a **time series plot** of the θ_j^* values;

(b) a **density trace** of the θ_j^* output, which is an **estimate** of $p(\theta_j|y)$;

(c) a plot of the **autocorrelation function** (ACF) of the θ_j^* values; and

The MCMC 4–Plot (continued)

(d) a plot of the **partial autocorrelation function** (PACF) of the θ_j^* output (see the **portrait version** of **chapter 3** of the **lecture notes** for **details** on the **ACF** and **PACF**).

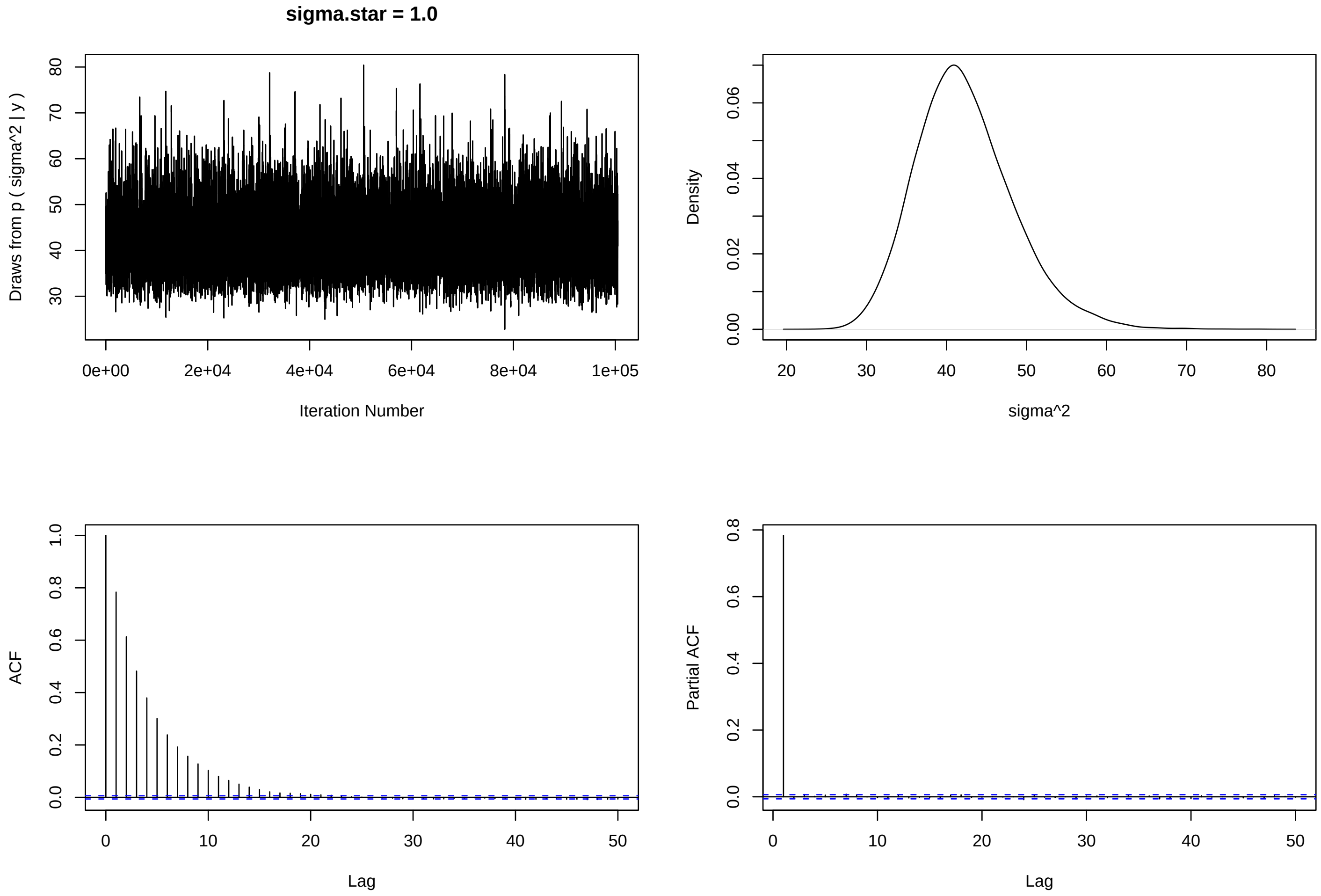
Here's some R code for the **MCMC 4–plot** for the **output** of the **sampler** with $\sigma_* = 1.0$:

```
par( mfrow = c( 2, 2 ) )
plot( 1:( M + B + 1 ), mcmc.data.set.1[ , 2 ], type = 'l',
  xlab = 'Iteration Number', ylab = 'Draws from p ( sigma^2 | y )',
  main = 'sigma.star = 1.0' )
plot( density( mcmc.data.set.1[ , 2 ], adjust = 2 ), xlab = 'sigma^2',
  ylab = 'Density', main = '' )
acf( mcmc.data.set.1[ , 2 ], main = '' )
pacf( mcmc.data.set.1[ , 2 ], main = '' )
```

The **plots** on the next page **reveal** the following:

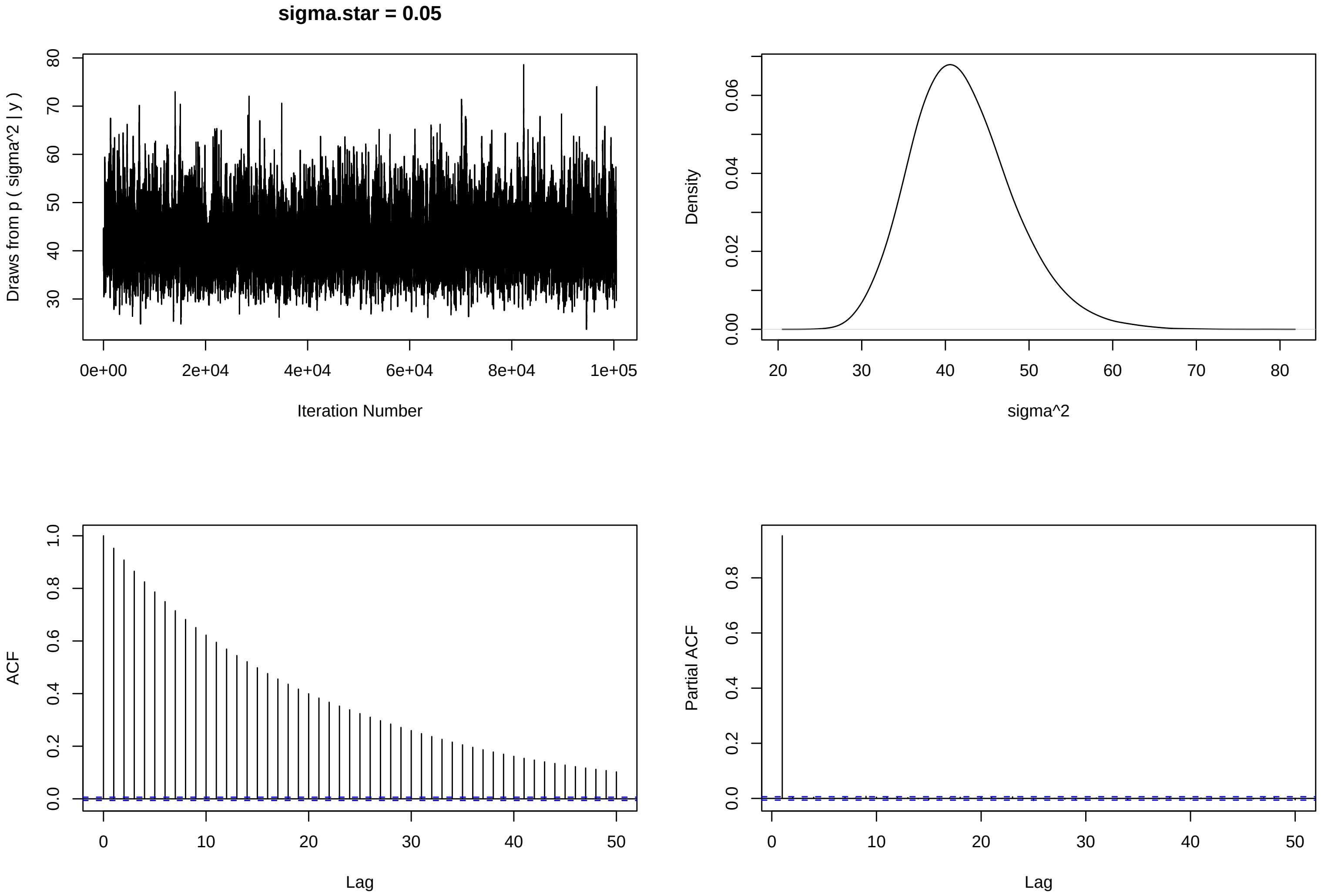
- the time series plot exhibits **stationarity**;
- the density trace looks like You would **expect** it to, for a **posterior distribution** for a **variance**;

The MCMC 4–Plot (continued)



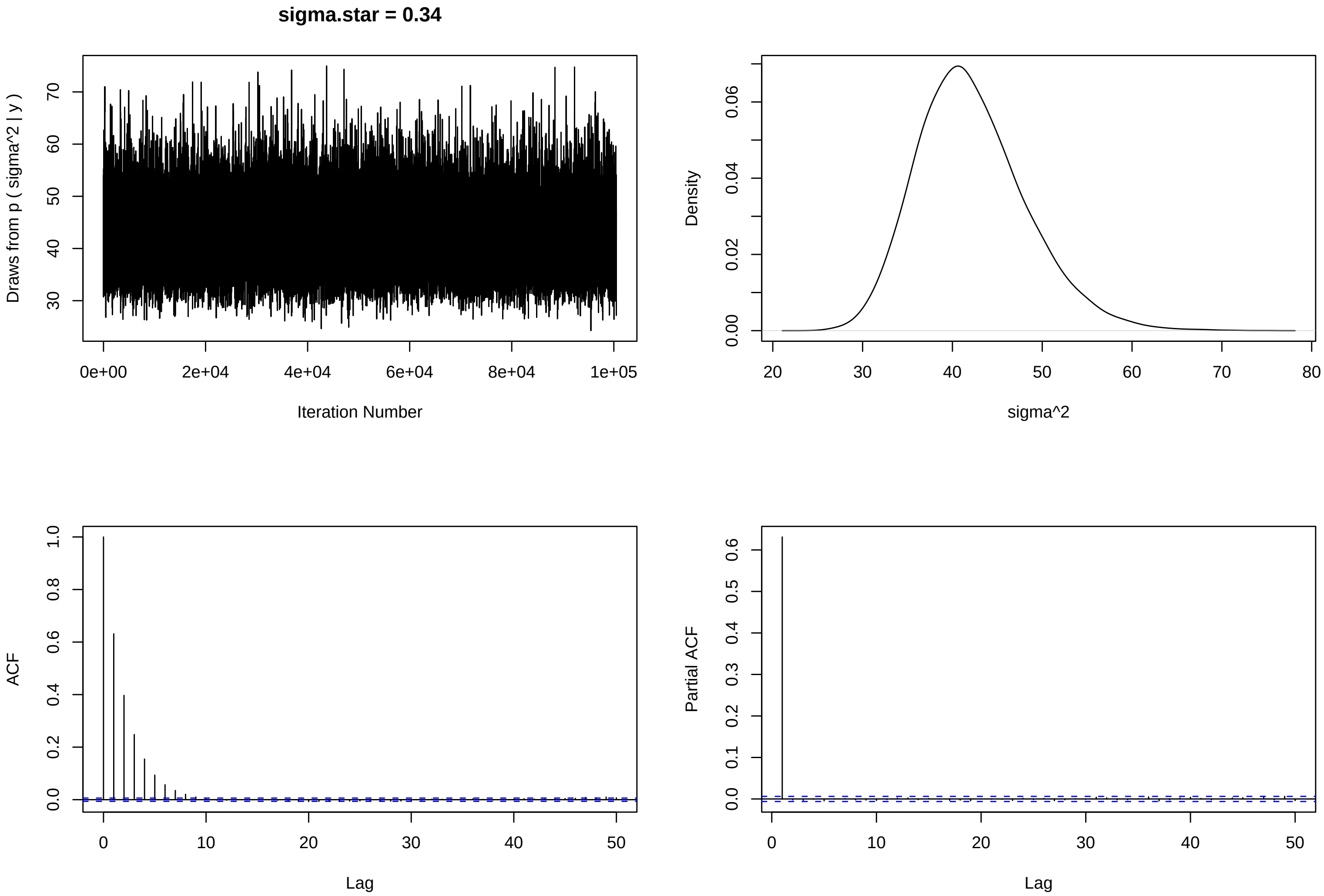
- the **PACF** diagnoses this **output** as that of an AR_1 process with a **first-order autocorrelation** of about $\hat{\rho}_1 \doteq 0.8$, and
- this is **consistent** with the **behavior** of the **ACF**.

Poor Mixing Shows Up in the 4–Plot



With $\sigma_* = 0.05$ the mixing is even worse than with $\sigma_* = 1.0$: the **first-order autocorrelation** is now about $\hat{\rho}_1 \doteq 0.95$.

Even the Optimal Sampler Has $\hat{\rho}_1 \doteq 0.6$



Even with the best-possible $\sigma_* \doteq 0.34$ (for this **random-walk Metropolis** sampler with a **Gaussian proposal** distribution on $\log \theta$), the **first-order autocorrelation** is still about $\hat{\rho}_1 \doteq 0.6$.

The Value of Fine Tuning

With **commands** like the following for **all three output series** —

```
print( rho.1 <- acf( mcmc.data.set.1[ , 2 ], plot = F )$acf[ 2 ] )
print( posterior.mean.1 <- mean( mcmc.data.set.1[ , 2 ] ) )
print( posterior.sd.1 <- sd( mcmc.data.set.1[ , 2 ] ) )
print( MCSE.1 <- ( posterior.sd.1 / sqrt( M ) ) *
  sqrt( ( 1 + rho.1 ) / ( 1 - rho.1 ) ) )
print( correct.posterior.mean <- sigma2.hat * n / ( n - 2 ) )
print( correct.posterior.sd <- sigma2.hat * sqrt( 2 * n^2 /
  ( ( n - 2 )^2 * ( n - 4 ) ) ) )
```

— you can **make** the following **table**:

	Acceptance	Posterior	MCSE of	Posterior
PDSD	Rate	Mean	Posterior Mean	SD
0.05	0.89	42.3	0.124	6.09
0.34	0.44	42.2	0.041	6.10
1.00	0.18	42.2	0.055	6.04

The correct **posterior mean and SD** are **42.25** and **6.097** (respectively).

Gibbs Sampling

Note that the **MCSE** for $\sigma_* = 0.05$ is $\frac{0.124}{0.041} \doteq 3.05$ times **bigger** than for $\sigma_* = 0.34$, so that the $\sigma_* = 0.05$ **sampler** would require $3.05^2 \doteq$ **9.3 times more monitoring iterations** to get the **same accuracy**.

As discussed in the **portrait version** of **chapter 3** of these **lecture notes**, when the goal is **summarization** (via **MCMC**) of a **posterior distribution** $p(\theta_1, \ldots, \theta_k | y)$ for $k > 1$, you have a fair amount of **flexibility**: you can **update all** the components of θ **simultaneously** with a k-dimensional **proposal distribution**, or you can update them **one at a time** (**single-scan MCMC**), or you can **block-update** some of them **simultaneously** and others **one at a time** (the **basic rule** is: always use the **most recent version** of anything that's **not currently being updated**).

Gibbs sampling is a **special case** of **Metropolis-Hastings sampling** in which (a) the **proposal distribution** for θ_j is the **full-conditional distribution** $p(\theta_j | \theta_{-j}, y)$ and (b) with this choice You **accept all proposed moves**; let's see an **example** in action.

Gibbs Sampling in the Gaussian Model

Gibbs sampling example. In the **Gaussian sampling model** with **both parameters unknown**, we saw in **Chapter 2** that the **conjugate model** was (for $i = 1, \ldots, n$)

$$\begin{array}{rcl} \sigma^2 & \sim & \text{SI-}\chi^2(\nu_0, \sigma_0^2) \\ (\mu | \sigma^2) & \sim & N\left(\mu_0, \dfrac{\sigma^2}{\kappa_0} \right) \\ (y_i | \mu, \sigma^2) & \stackrel{\text{IID}}{\sim} & N(\mu, \sigma^2) \, , \end{array} \tag{24}$$

and the **full conditionals** for this model were shown (in the **portrait version** of **Chapter 3**) to be

$$\begin{array}{rcl} (\mu | \sigma^2, y) & \sim & N\left(\dfrac{\kappa_0 \, \mu_0 + n \, \bar{y}}{\kappa_0 + n}, \dfrac{\sigma^2}{\kappa_0 + n} \right) \quad \text{and} \\ (\sigma^2 | \mu, y) & \sim & \text{SI-}\chi^2\left(\nu_0 + 1 + n, \dfrac{\nu_0 \, \sigma_0^2 + \kappa_0 (\mu - \mu_0)^2 + n \, s_\mu^2}{\nu_0 + 1 + n} \right) , \end{array} \tag{25}$$

in which $\bar{y} = \frac{1}{n} \sum_{i=1}^n y_i$ and $s_\mu^2 = \frac{1}{n} \sum_{i=1}^n (y_i - \mu)^2$.

Gibbs Sampling in the Gaussian Model (continued)

We **already know** (Chapter 2) that the **correct marginal posteriors** for μ and σ^2 and the **correct posterior predictive distribution** for the **next observation** are

$$\begin{aligned} (\sigma^2|y) &\sim \text{SI-}\chi^2(\nu_n, \sigma_n^2), \\ (\mu|y) &\sim t_{\nu_n}\left(\mu_n, \frac{\sigma_n^2}{\kappa_n}\right), \quad \text{and} \\ (y_{n+1}|y) &\sim t_{\nu_n}\left(\mu_n, \frac{\kappa_n+1}{\kappa_n}\sigma_n^2\right), \quad \text{in which} \end{aligned} \tag{26}$$

$$\begin{aligned} \nu_n &= \nu_0 + n, \\ \sigma_n^2 &= \frac{1}{\nu_n}\left[\nu_0\sigma_0^2 + (n-1)s^2 + \frac{\kappa_0 n}{\kappa_0+n}(\bar{y}-\mu_0)^2\right], \\ \mu_n &= \frac{\kappa_0}{\kappa_0+n}\mu_0 + \frac{n}{\kappa_0+n}\bar{y}, \quad \text{and} \\ \kappa_n &= \kappa_0 + n, \end{aligned} \tag{27}$$

with $s^2 = \frac{1}{n-1}\sum_{i=1}^n (y_i - \bar{y})^2$ as the **usual sample variance**.

Here's some `R` code to **implement** the **Gibbs sampler** in this problem:

R Implementation

```
gibbs.example <- function( y, nu.0, sigma2.0, mu.0, kappa.0, mu.initial,
    sigma2.initial, M, B, seed ) {
  set.seed( seed )
  n <- length( y )
  y.bar <- mean( y )
  mcmc.data.set <- matrix( NA, M + B + 1, 4 )
  mcmc.data.set[ 1, ] <- c( mu.initial, sigma2.initial,
    sqrt( sigma2.initial ), mu.initial )
  for ( i in 2:( M + B + 1 ) ) {
    mu.star <- rnorm( 1, ( kappa.0 * mu.0 + n * y.bar ) / ( kappa.0 + n ),
      sqrt( mcmc.data.set[ i - 1, 2 ] / ( kappa.0 + n ) ) )
    s2.mu.star <- sum( ( y - mu.star )^2 ) / n
    sigma2.star <- rsichi2( 1, nu.0 + 1 + n, ( nu.0 * sigma2.0 +
      kappa.0 * ( mu.star - mu.0 )^2 + n * s2.mu.star ) /
      ( nu.0 + 1 + n ) )
    y.star <- rnorm( 1, mu.star, sqrt( sigma2.star ) )
    mcmc.data.set[ i, ] <- c( mu.star, sigma2.star,
      sqrt( sigma2.star ), y.star )
    if ( ( i %% 1000 ) == 0 ) print( i )
  }
```

Simulating From the Posterior Predictive Distribution

```
  return( mcmc.data.set )
}
rsichi2 <- function( n, nu, s2 ) {
  return( nu * s2 / rchisq( n, nu ) )
}
```

There are **three new things** about this **code** in relation to the previous **random-walk-Metropolis** example: (a) **sampling** from the **full-conditional distributions** (Gibbs); (b) **monitoring** a function of $\theta = (\mu, \sigma^2)$ (in this case, $\sigma = \sqrt{\sigma^2}$); and (c) **sampling** from the **posterior predictive distribution** $p(y_{n+1}|y)$ (we could have done (b) and (c) **in the same way** in the previous example; I just **forgot** to do so), which You'll **recall** has the form

$$p(y_{n+1}|y) = \int p(y_{n+1}|\theta)\, p(\theta|y)\, d\theta\,. \tag{28}$$

Equation (28) represents $p(y_{n+1}|y)$ **hierarchically** as a **mixture** of $p(y_{n+1}|\theta)$ weighted by $p(\theta|y)$; back in Chapter 2 we agreed that to **make a random draw** y^*_{n+1} from such a **mixture** you just have to (i) draw θ from $p(\theta|y)$, obtaining θ^*, and then (ii) draw y_{n+1} from $p(y_{n+1}|\theta^*)$.

Testing the Code

But that's **exactly** what the line

```
y.star <- rnorm( 1, mu.star, sqrt( sigma2.star ) )
```

does, to fill in a **draw** y^*_{n+1} from $p(y_{n+1}|y)$ in **each row** of the **MCMC data set**.

To test this code I generated a little **artificial data set**, made the **following choices** for the **prior** and other inputs to the **driver function**, and ran it:

```
print( sort( signif( rnorm( 10, 50, 5 ), 3 ) ) )
# 42.1 44.0 44.7 48.9 48.9 52.5 54.0 55.4 58.2 59.0
y <- c( 42.1, 44.0, 44.7, 48.9, 48.9, 52.5, 54.0, 55.4, 58.2, 59.0 )
mean( y )
# 50.77
sd( y )
# 5.987403
nu.0 <- 10
sigma2.0 <- 15
mu.0 <- 90
kappa.0 <- 10
mu.initial <- 70
```

Checking the Results

```
sigma2.initial <- 20
M <- 99999
B <- 0
seed <- c( 123456, 654321 )
mcmc.data.set.1 <- gibbs.example( y, nu.0, sigma2.0, mu.0, kappa.0,
  mu.initial, sigma2.initial, M, B, seed )
```

This took about **20 seconds** at **1.6 Unix GHz**; to **check the results** I can **simulate draws** from the **correct marginal posterior distributions** for μ and σ^2 and the correct **posterior predictive distribution** for y_{n+1}, and **compare** with the **MCMC output**:

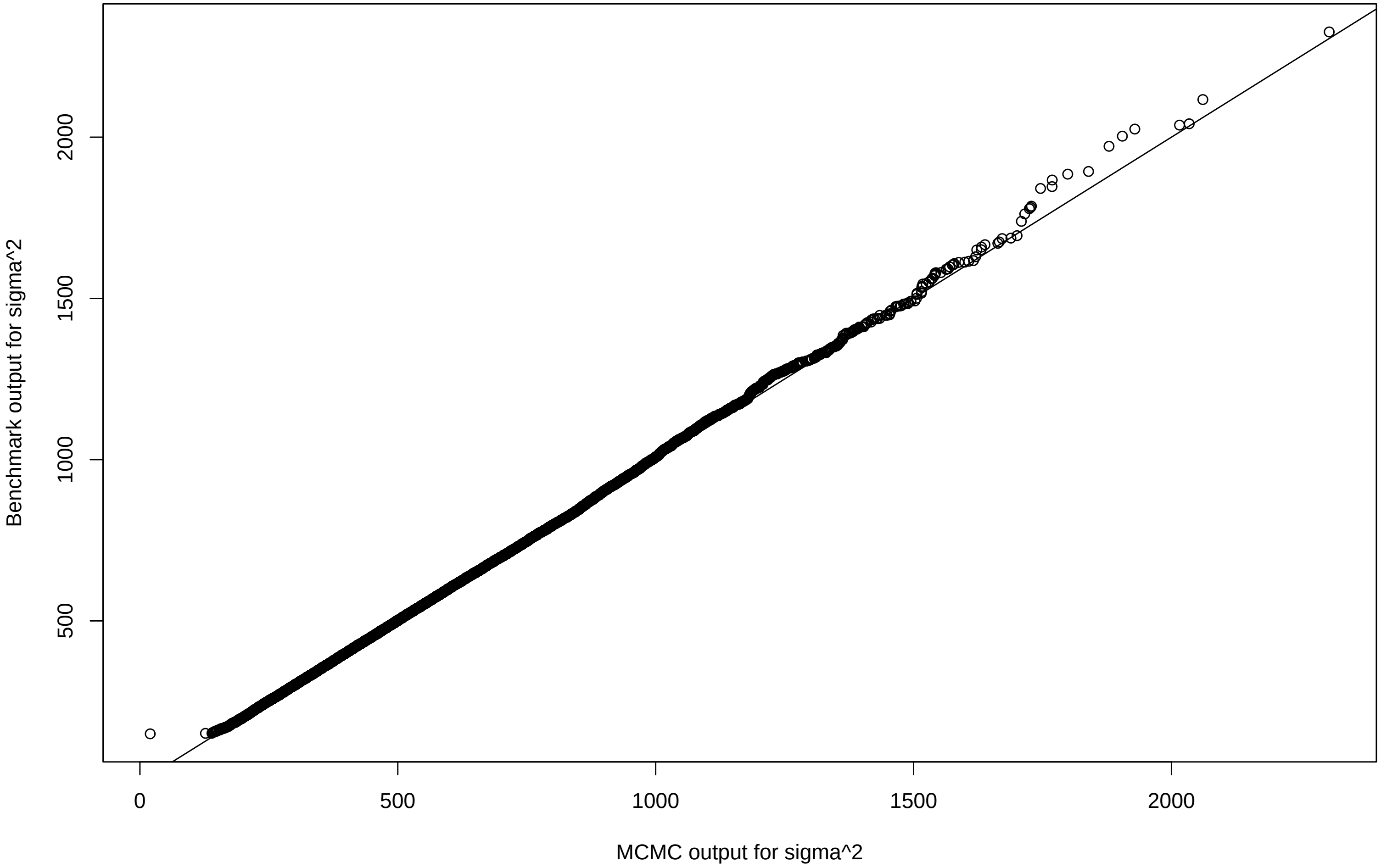
```
print( n <- length( y ) )
# 10
print( nu.n <- nu.0 + n )
# 20
print( y.bar <- mean( y ) )
# 50.77
print( s2 <- var( y ) )
# 35.849
```

Checking the Results (continued)

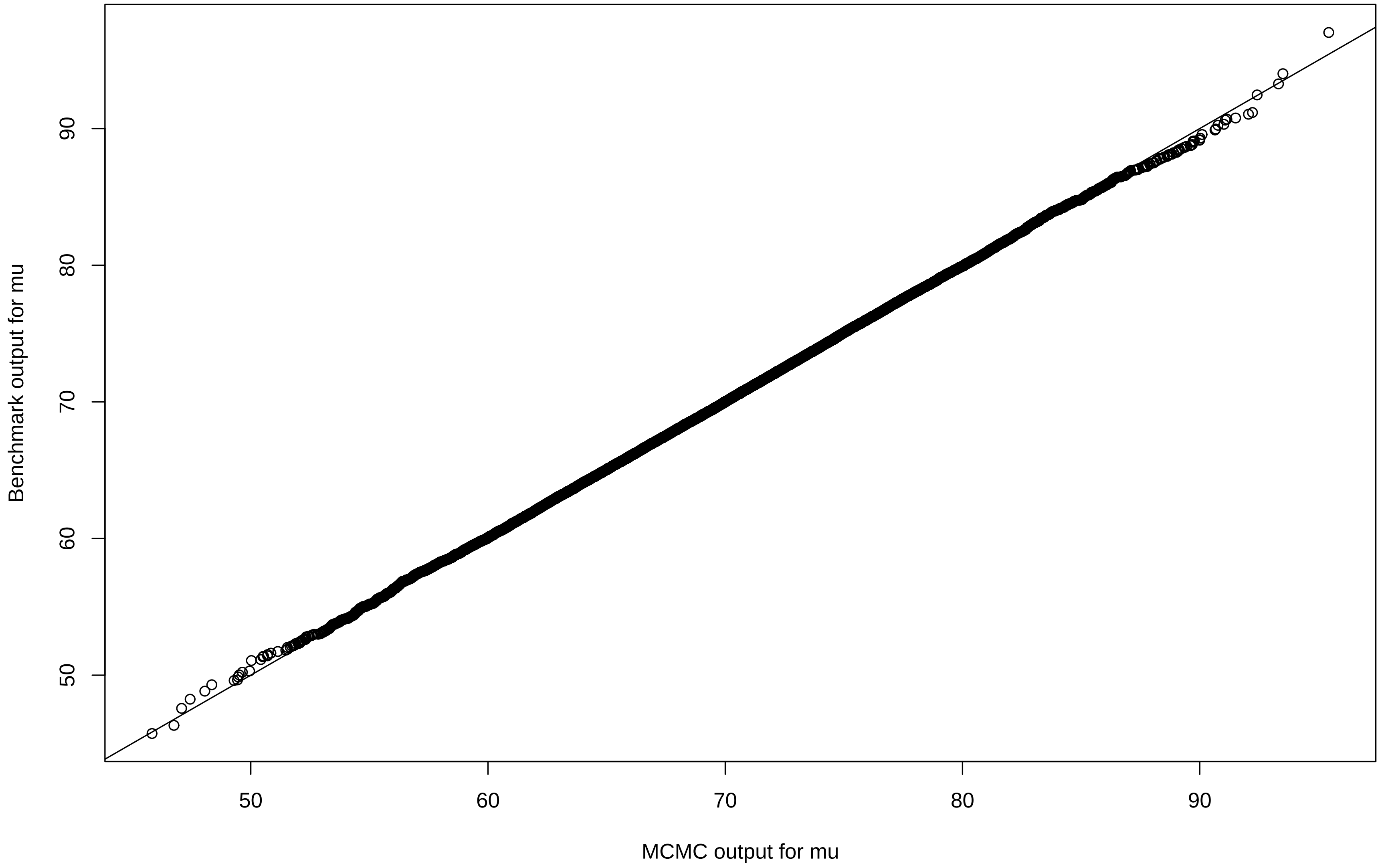
```
print( sigma2.n <- ( nu.0 * sigma2.0 + ( n - 1 ) * s2 + kappa.0 * n *
  ( y.bar - mu.0 )^2 / ( kappa.0 + n ) ) / nu.n )
# 408.3803
print( mu.n <- ( kappa.0 * mu.0 + n * y.bar ) / ( kappa.0 + n ) )
# 70.385
print( kappa.n <- kappa.0 + n )
# 20
sigma2.benchmark <- rsichi2( M + B + 1, nu.n, sigma2.n )
qqplot( mcmc.data.set.1[ , 2 ], sigma2.benchmark,
  xlab = ’MCMC output for sigma^2’, ylab = ’Benchmark output for sigma^2’ )
abline( 0, 1 )
rscaled.t <- function( n, mu, sigma2, nu ) {
  return( sqrt( sigma2 ) * rt( n, nu ) + mu )
}
mu.benchmark <- rscaled.t( M + B + 1, mu.n, sigma2.n / kappa.n, nu.n )
qqplot( mcmc.data.set.1[ , 1 ], mu.benchmark,
  xlab = ’MCMC output for mu’, ylab = ’Benchmark output for mu’ )
abline( 0, 1 )
y.next.benchmark <- rscaled.t( M + B + 1, mu.n, ( kappa.n + 1 ) *
  sigma2.n / kappa.n, nu.n )
```

Checking the Results (continued)

```
qqplot( mcmc.data.set.1[ , 4 ], y.next.benchmark,
  xlab = 'MCMC output for y.next', ylab = 'Benchmark output for y.next' )
abline( 0, 1 )
```

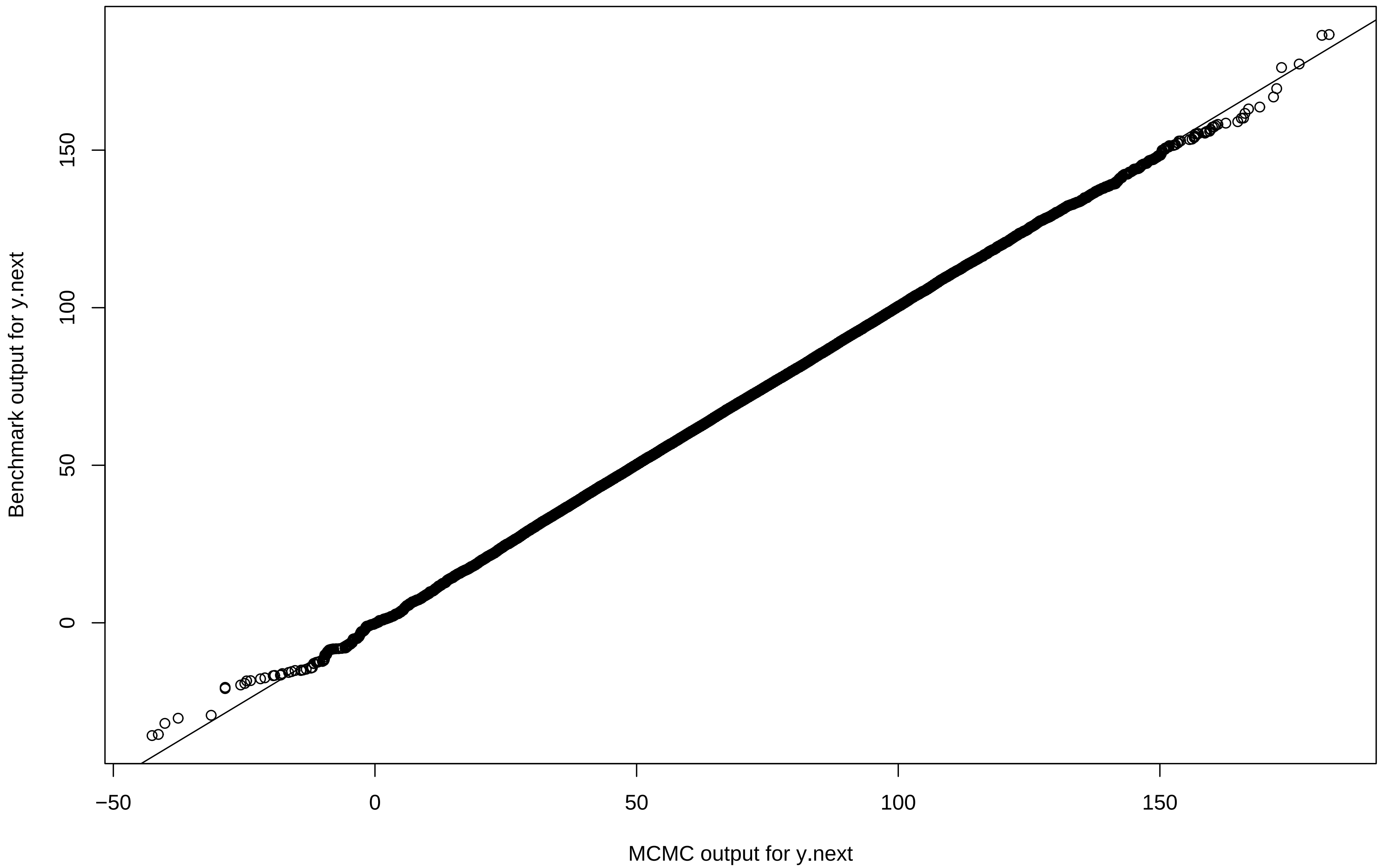


Checking the Results (continued)



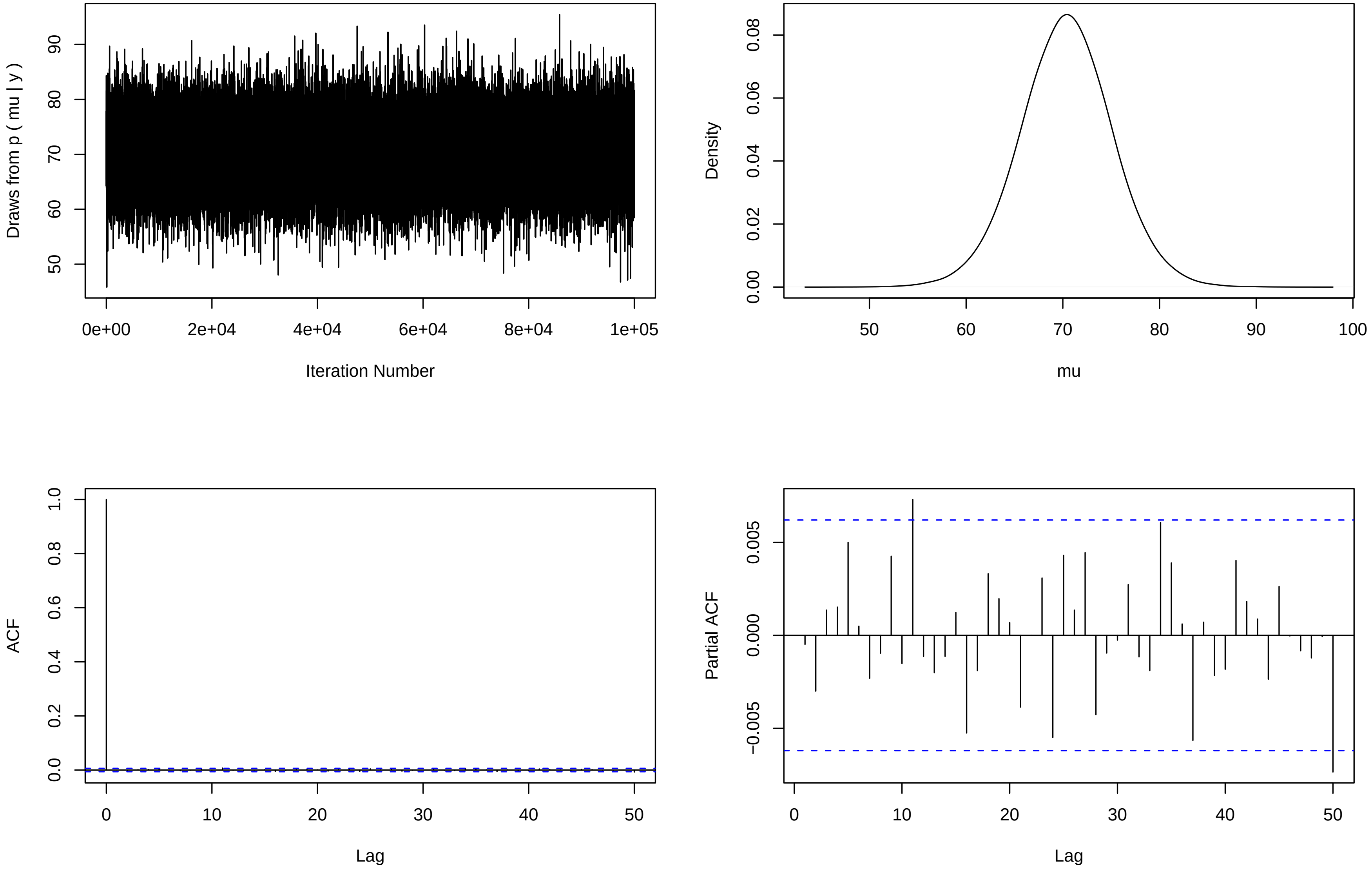
All of these **plots** (on pages 64-66) **look great**.

Checking the Results (continued)



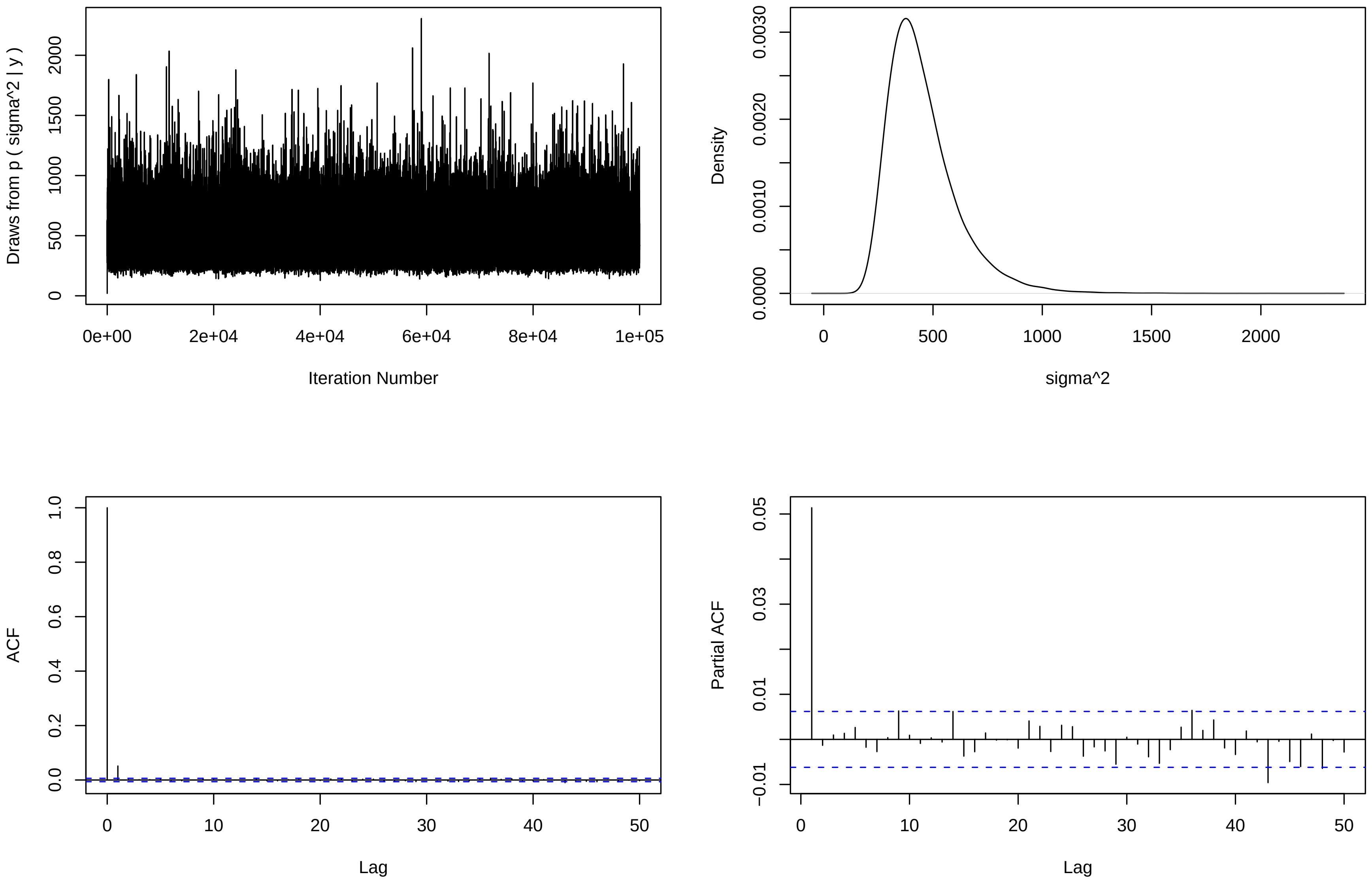
So now we can **begin looking** at the **MCMC 4–plots**:

MCMC 4–Plot for μ

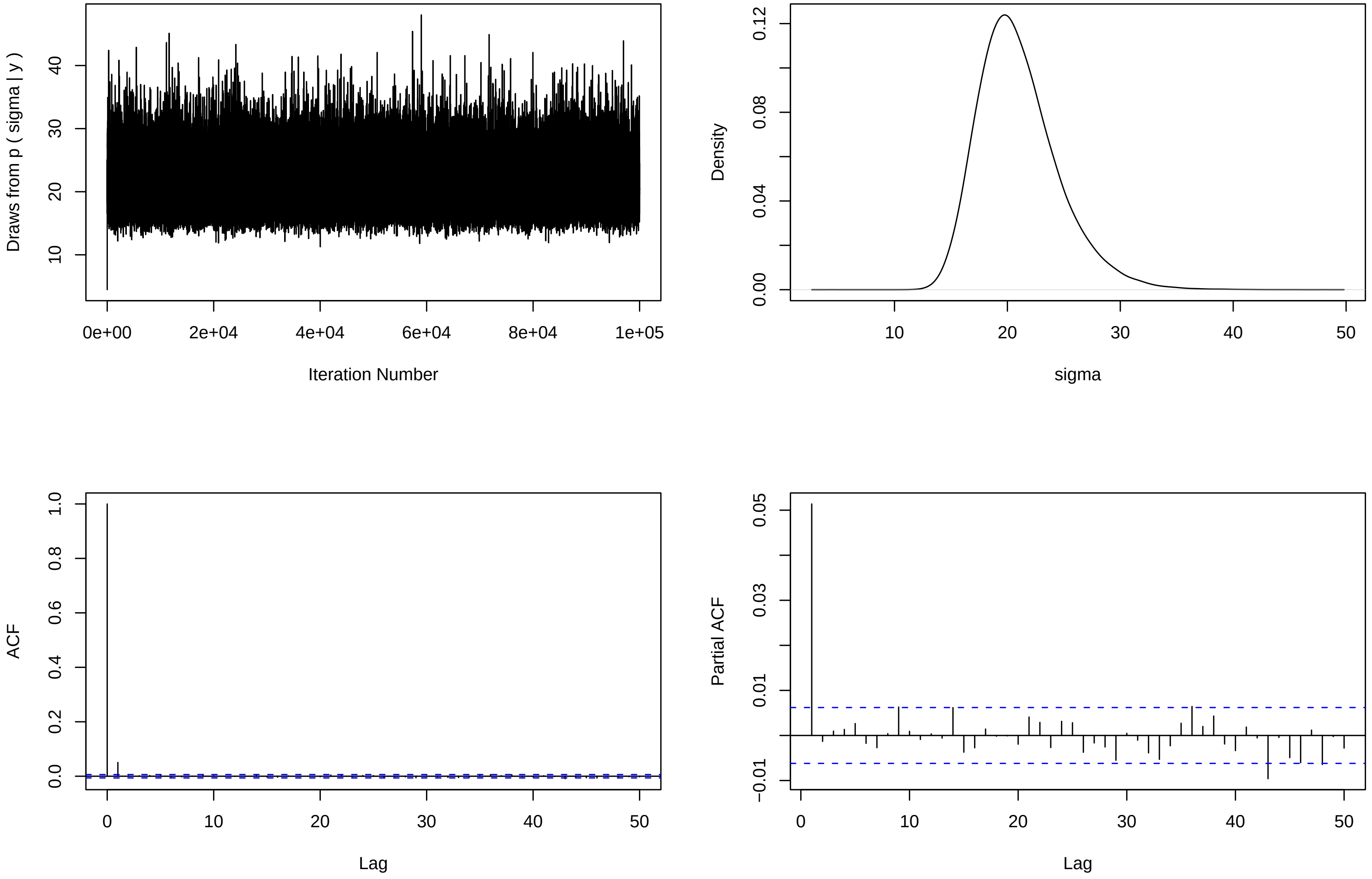


In this problem **Gibbs sampling $\doteq$ IID sampling**.

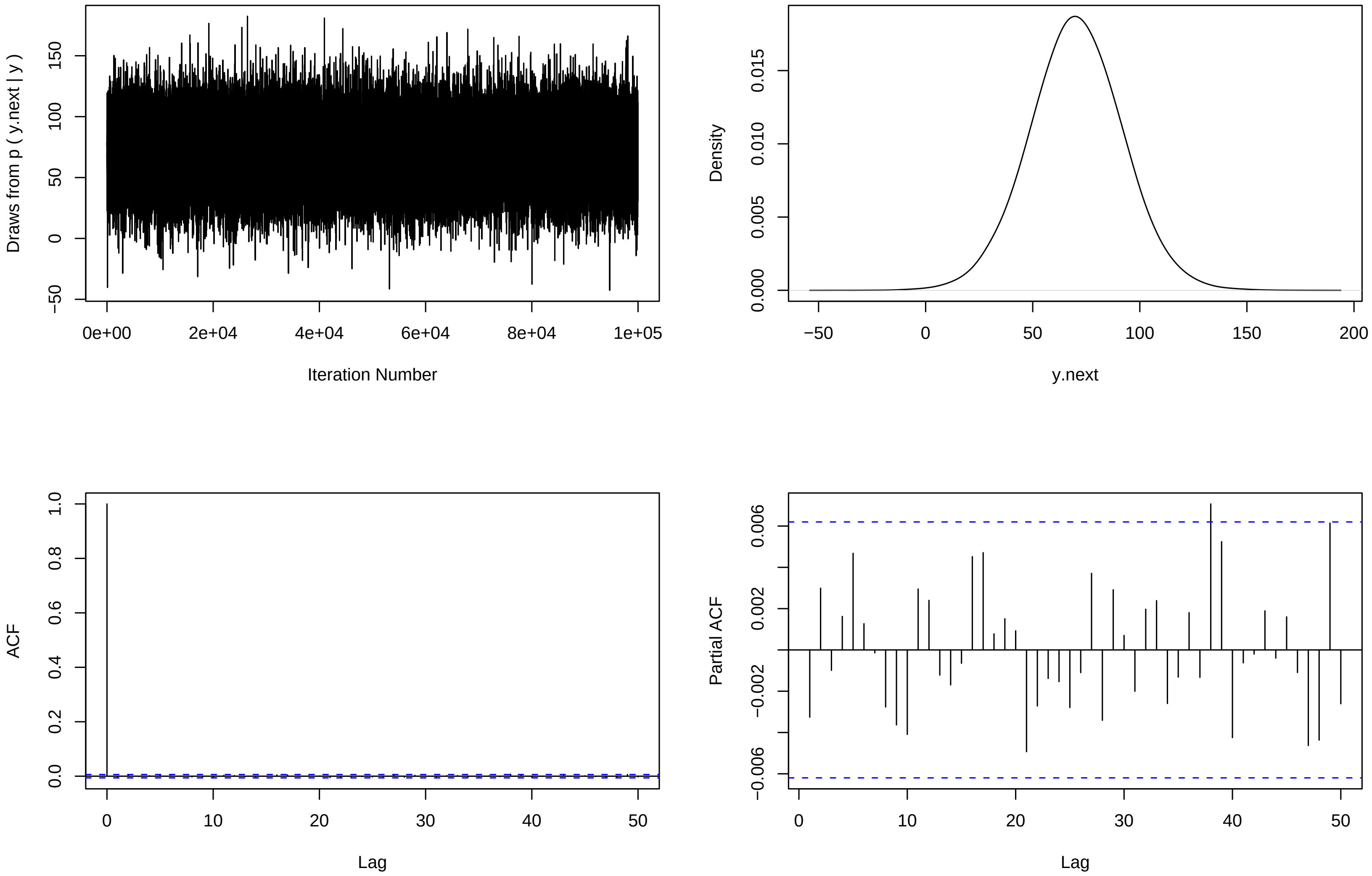
MCMC 4–Plot for σ^2



MCMC 4–Plot for σ



MCMC 4–Plot for y_{n+1}



With the hierarchical sampling method, the **draws** from $p(y_{n+1}|y)$ are **always IID**.

Numerical Posterior Summaries

With **commands** like those on **page 55** you can readily produce the following **numerical posterior summaries**:

Unknown	Posterior Mean	MCSE of Posterior Mean	Posterior SD	Quantiles 2.5%	97.5%
μ	70.37	0.015	4.766	60.89	79.82
σ^2	453.8	0.509	161.1	238.4	856.8
σ	21.01	0.011	3.541	15.44	29.27
y_{n+1}	70.30	0.069	21.79	27.41	113.3

You can **see** that

(a) **100,000 monitoring iterations** (about **20 seconds** of CPU time) has achieved **about 4 significant figures** worth of accuracy in the **posterior means** (and therefore **roughly the same order of magnitude** of accuracy for the **other summaries**), and

(b) (as usual) the **posterior uncertainty** about the **next observation** is **substantially bigger** than the **uncertainty** about the **mean** of the **process**.

NB10 Case Study

We're now ready to do **MCMC sampling** in the **NB10 case study**, for which the **model** is (for $i = 1, \ldots, n$)

$$\begin{aligned} (\mu, \sigma^2, \nu) & \sim & p(\mu, \sigma^2, \nu) \\ (y_i \vert \mu, \sigma^2, \nu) & \stackrel{\text{IID}}{\sim} & t_\nu(\mu, \sigma^2) \, . \end{aligned} \qquad (29)$$

The **sampling distribution/likelihood function** in this **model** is

$$\begin{aligned} p(y \vert \mu, \sigma^2, \nu) & = & \prod_{i=1}^n \frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\Gamma\left(\frac{\nu}{2}\right) \sqrt{\pi \, \nu \, \sigma^2}} \left[1 + \frac{(y_i - \mu)^2}{\nu \, \sigma^2} \right]^{-\frac{\nu+1}{2}} = \qquad (30) \\ l(\mu, \sigma^2, \nu \vert y) & = & c \frac{\left[\Gamma\left(\frac{\nu+1}{2}\right)\right]^n}{\left[\Gamma\left(\frac{\nu}{2}\right)\right]^n} \nu^{-\frac{n}{2}} \left(\sigma^2\right)^{-\frac{n}{2}} \left\{ \prod_{i=1}^n \left[1 + \frac{(y_i - \mu)^2}{\nu \, \sigma^2} \right] \right\}^{-\frac{\nu+1}{2}} ; \end{aligned}$$

this leads to the **log likelihood function**

$$\begin{aligned} ll(\mu, \sigma^2, \nu \vert y) & = & n \log \Gamma \left(\frac{\nu+1}{2} \right) - n \log \Gamma \left(\frac{\nu}{2} \right) - \frac{n}{2} \log \nu - \frac{n}{2} \log \sigma^2 - \\ & & \left(\frac{\nu+1}{2} \right) \sum_{i=1}^n \log \left[1 + \frac{(y_i - \mu)^2}{\nu \, \sigma^2} \right] . \qquad (31) \end{aligned}$$

Generic Random-Walk Metropolis

Gibbs sampling would be **no fun** in this **model**; it turns out that a good generic MCMC approach is **single-scan random-walk Metropolis** with **Gaussian proposal distributions** on each of the **components** of the **vector** θ of **unknowns**, after **all components** have been **transformed** to **live** on the **entire real line**.

So define $\eta = \log \sigma^2$ and $\gamma = \log \nu$, so that $\theta = (\mu, \eta, \gamma)$ with $\sigma^2 = e^\eta$ and $\nu = e^\gamma$; in this **parameterization** the **log likelihood function** is

$$\begin{aligned} ll(\mu, \eta, \gamma | y) \quad = \quad & n \, \log \Gamma \left(\frac{e^\gamma + 1}{2} \right) - n \, \log \Gamma \left(\frac{e^\gamma}{2} \right) - \frac{n}{2} \gamma - \frac{n}{2} \eta - \\ & \left(\frac{e^\gamma + 1}{2} \right) \sum_{i=1}^n \log \left[1 + \frac{(y_i - \mu)^2}{e^{\gamma + \eta}} \right] . \qquad (32) \end{aligned}$$

If **context** implies a **diffuse prior** on (μ, σ^2, ν) — as it does in the **NB10 case study** — then a **reasonable place to start** would be with the **improper prior** defined by $\log p(\mu) = \log p(\eta) = \log p(\gamma) = 0$; we can then do **sensitivity analysis** to see how much **this particular diffuse prior choice** affects the **posterior**.

Exploring the Log Likelihood

Before we launch into the **MCMC**, let's take a bit of time to **explore the log likelihood surface**, first in `R` and then in `Maple`.

```
log.likelihood <- function( theta, y ) {
  n <- length( y )
  mu <- theta[ 1 ]
  eta <- theta[ 2 ]
  gamma <- theta[ 3 ]
  return( n * lgamma( ( exp( gamma ) + 1 ) / 2 ) - n *
    lgamma( exp( gamma ) / 2 ) - n * gamma / 2 - n * eta / 2 -
    ( ( exp( gamma ) + 1 ) / 2 ) * sum( log( 1 + ( y - mu )^2 /
    exp( gamma + eta ) ) ) )
}
```

`R` has a **built-in function** called `optim` that's good at **minimizing scalar functions of vector arguments** without having to supply **derivative information**; however, the **function** I give it to optimize should depend only on θ (not on θ and y), and to use it I need to **minimize minus the log likelihood**, so I'll define y as a **variable** in the **workspace** (rather than **passing** it into the function as an

Finding the MLE Vector

argument) and **rewrite** the function a bit:

```
y <- c( 409., 400., 406., 399., 402., 406., 401., 403., 401., 403., 398.,
  403., 407., 402., 401., 399., 400., 401., 405., 402., 408., 399., 399.,
     .
     .
     .
  412., 393., 437., 418., 415., 404., 401., 401., 407., 412., 375., 409.,
  406., 398., 406., 403., 404. )
minus.log.likelihood.for.optim <- function( theta ) {
  n <- length( y )
  mu <- theta[ 1 ]
  eta <- theta[ 2 ]
  gamma <- theta[ 3 ]
  log.likelihood <- n * lgamma( ( exp( gamma ) + 1 ) / 2 ) - n *
    lgamma( exp( gamma ) / 2 ) - n * gamma / 2 - n * eta / 2 -
    ( ( exp( gamma ) + 1 ) / 2 ) * sum( log( 1 + ( y - mu )^2 /
    exp( gamma + eta ) ) )
  return( - log.likelihood )
}
```

Finding the MLE Vector (continued)

`optim` requires a **vector** of **starting values** for its optimization search: I'll start μ off at the **sample mean** of the y values (which should not be far from the **MLE** for μ in the t model); I'll start $\eta = \log \sigma^2$ off at the **log of the sample variance** of the y values (σ^2 is **not quite the variance** of y_i in this model, but it's **not far off**); and for ν, I'll remember that **Churchill Eisenhart** (an old **NBS** hand) once said that **all measurement processes are approximately** t_8 and start $\gamma = \log \nu$ off at $\log 8$:

```
print( mle <- optim( c( mean( y ), log( var( y ) ), log( 8 ) ),
  minus.log.likelihood.for.optim, hessian = T ) )
$par
[1] 404.278968   2.617090   1.101340
$value
[1] 250.8514
$counts
function gradient
     106       NA
```

The MLE and the Hessian Matrix

```
$convergence
[1] 0
$message
NULL
$hessian
           [,1]        [,2]        [,3]
[1,]  4.7201671  -0.3438449  -0.3419223
[2,] -0.3438449  26.5910742 -10.9292142
[3,] -0.3419223 -10.9292142  16.7254706
```

This **output** means that (a) **convergence** was **successful** (`$convergence = 0`) after **106 function evaluations**; (b) the **maximum log likelihood value** was -250.8512, attained at the **MLE vector** $(\hat{\mu}, \hat{\eta}, \hat{\gamma}) \doteq (404.3, 2.617, 1.101)$; and (c) the **Hessian** of **minus the log likelihood** at the **MLE** is the `$hessian` **matrix** above.

`optim` may be **sensitive** to the **quality** of its **starting values**:

```
optim( c( 0, 0, 0 ), minus.log.likelihood.for.optim, hessian = T )
$par
[1] 11.23798 11.93484 30.61009
```

An Example of Failure to Converge

```
$value
[1] 677.9946
$counts
function gradient
     172       NA
$convergence
[1] 10
$message
NULL
$hessian
     [,1]     [,2]        [,3]
[1,]    0    0.000       0.000
[2,]    0 2183.744    2135.702
[3,]    0 2135.702 1002138.411
```

The **documentation** says that "convergence = 10 indicates **degeneracy of the Nelder-Mead simplex**," and of course **nothing** in any of this **guarantees** that it has found the global minimum (even with the **good starting values** I gave it earlier).

Large-Sample MLE-Based Interval Estimates

You'll recall that in **repeated sampling** the **large-sample distribution** of the **MLE vector** $\hat{\theta}$ is **approximately** $N_k\left(\theta, \hat{\Sigma}\right)$ with $k = 3$ in this example and $\hat{\Sigma} = \hat{I}^{-1}$, where $\hat{I}$ is **minus the Hessian matrix** of the **log likelihood function** evaluated at $\hat{\theta}$ (which equals the **Hessian matrix** of **minus the log likelihood** at the **same point**); so to get $\hat{\Sigma}$ I just have to **invert** the **Hessian**:

```
print( Sigma.theta.hat <- solve( mle$hessian ) )
            [,1]        [,2]       [,3]
[1,] 0.212918550 0.006210111 0.00841072
[2,] 0.006210111 0.051596574 0.03384260
[3,] 0.008410719 0.033842598 0.08207535
```

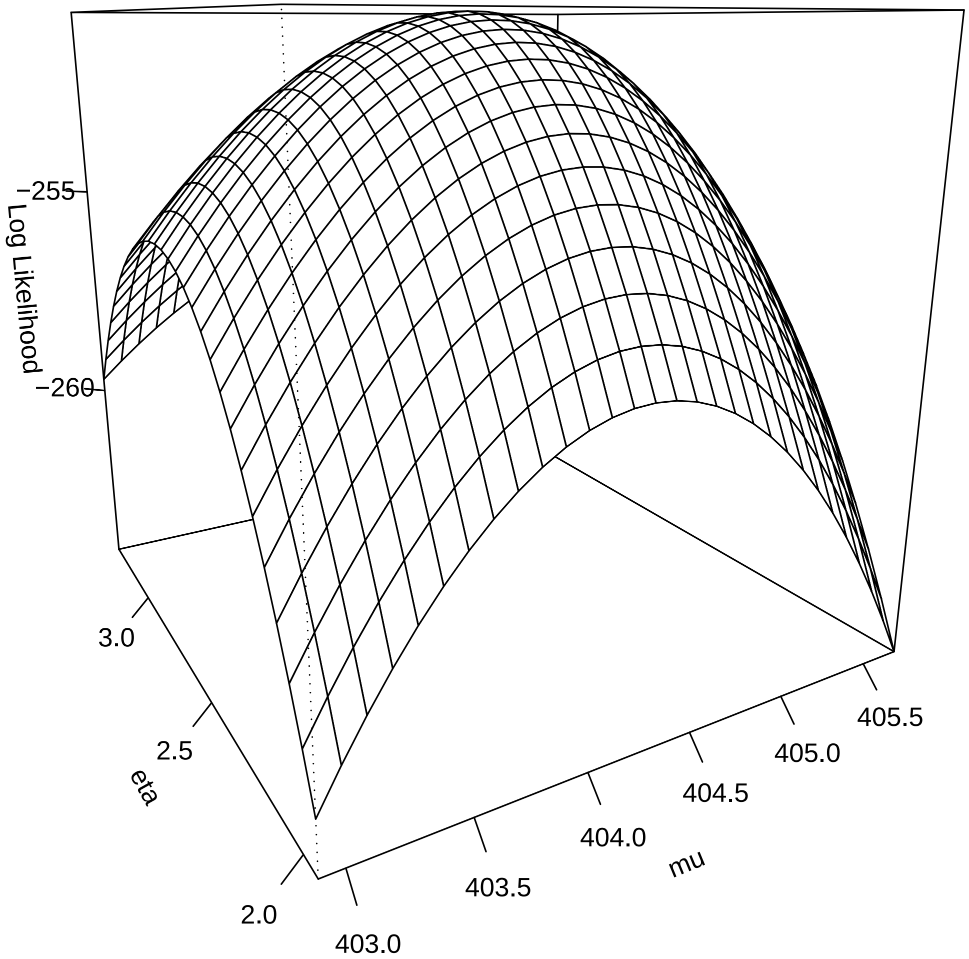
So this means that an **approximate 99.7% interval estimate** for component j of θ is $\hat{\theta}_j \pm 3\sqrt{\hat{\Sigma}_{jj}}$, where $\hat{\Sigma}_{ij}$ is the (i, j) **element** of $\hat{\Sigma}$; thus, according to the **likelihood approach**, μ is likely to be in the **interval** $404.3 \pm 3\sqrt{0.2129} \doteq (402.9, 405.7)$, and similarly the **99.7% intervals** for η and γ are **approximately** $(1.936, 3.298)$ and $(0.2415, 1.960)$, respectively.

Exploring the Log Likelihood Surface in R

Armed with this **information** we can **explore** the log likelihood surface near its **maximum**, by **holding one component** of θ **constant** at its **MLE** and making **3d-perspective** and **contour plots** as a **function** of the **other two components**:

```
theta.hat <- c( 404.278968, 2.617090, 1.101340 )
n.grid <- 30
mu.grid <- seq( 402.9, 405.7, length = n.grid )
eta.grid <- seq( 1.936, 3.298, length = n.grid )
gamma.grid <- seq( 0.2415, 1.960, length = n.grid )
mu.eta.log.likelihood.grid <- array( data = NA, dim = c( n.grid, n.grid ) )
for ( i in 1:n.grid ) {
  for ( j in 1:n.grid ) {
    mu.eta.log.likelihood.grid[ i, j ] <- log.likelihood( c( mu.grid[ i ],
      eta.grid[ j ], theta.hat[ 3 ] ), y )
  }
}
persp( mu.grid, eta.grid, mu.eta.log.likelihood.grid, xlab = 'mu',
  ylab = 'eta', zlab = 'Log Likelihood', axes = T,
  ticktype = 'detailed', theta = -30, phi = 15 )
```

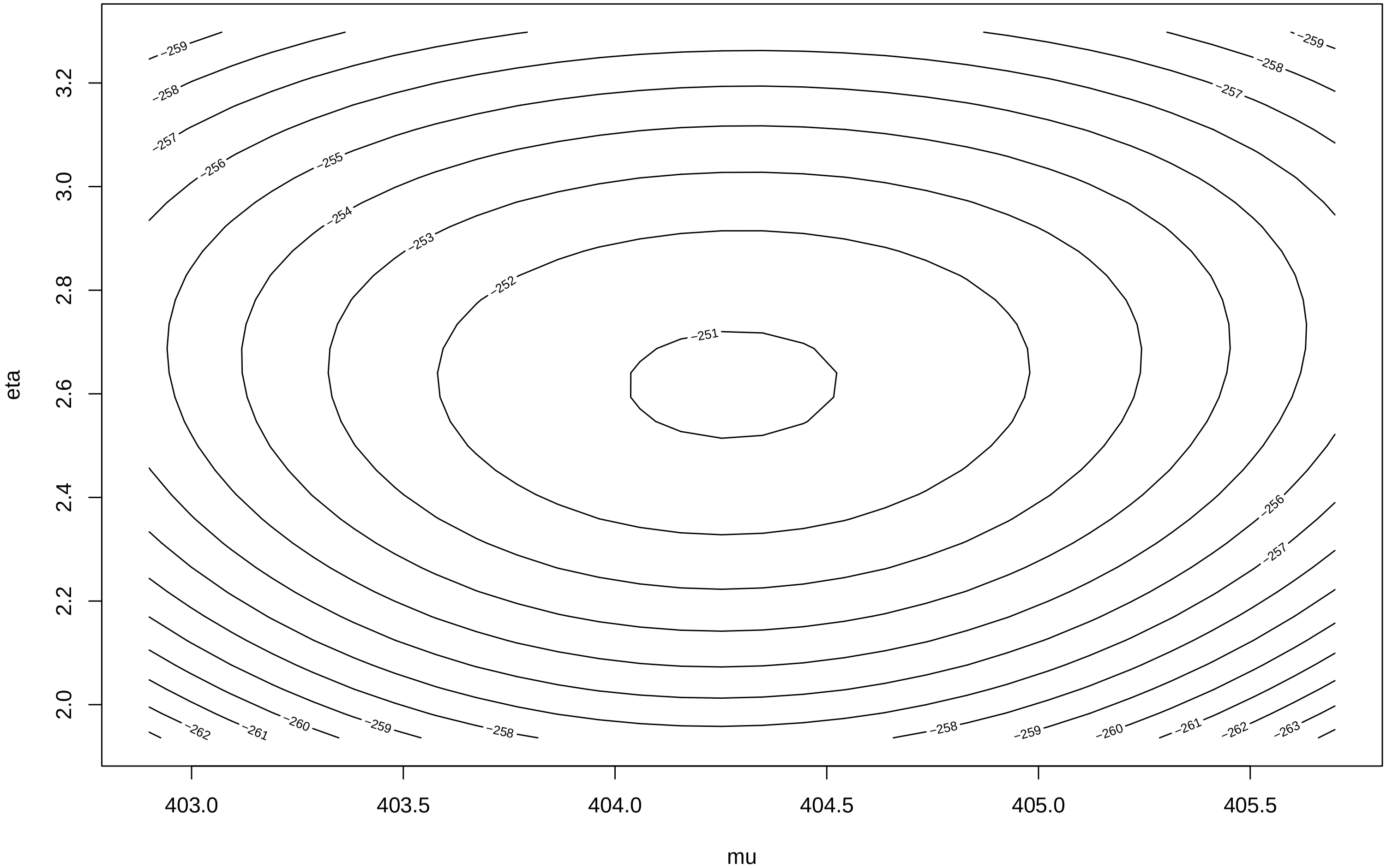
The Log Likelihood Surface in (μ, η) (3D-Perspective Plot)



This looks **pleasingly bivariate normal** in μ and η, with a **well-defined (global) maximum**.

The Log Likelihood Surface in (μ, η) (Contour Plot)

```
contour( mu.grid, eta.grid, mu.eta.log.likelihood.grid, xlab = 'mu',
  ylab = 'eta' )
```



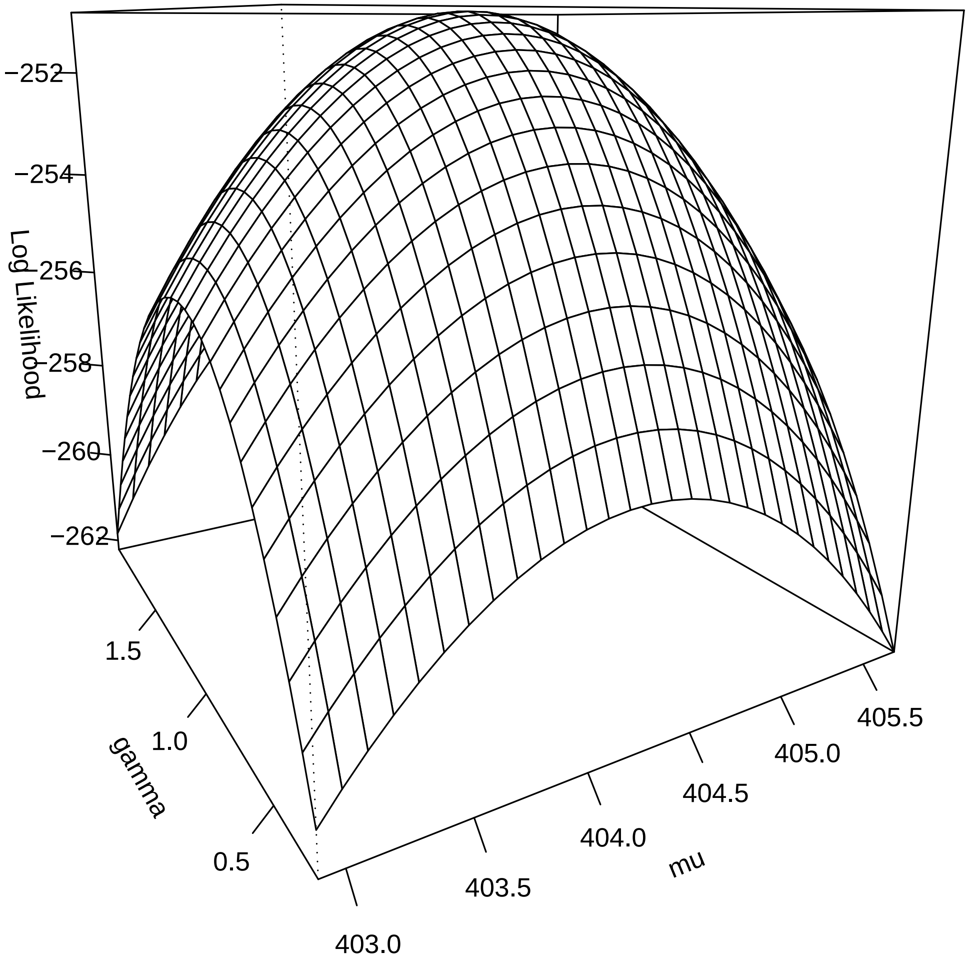
μ and η will be **uncorrelated** in the **posterior**.

The Log Likelihood Surface in (μ, γ)

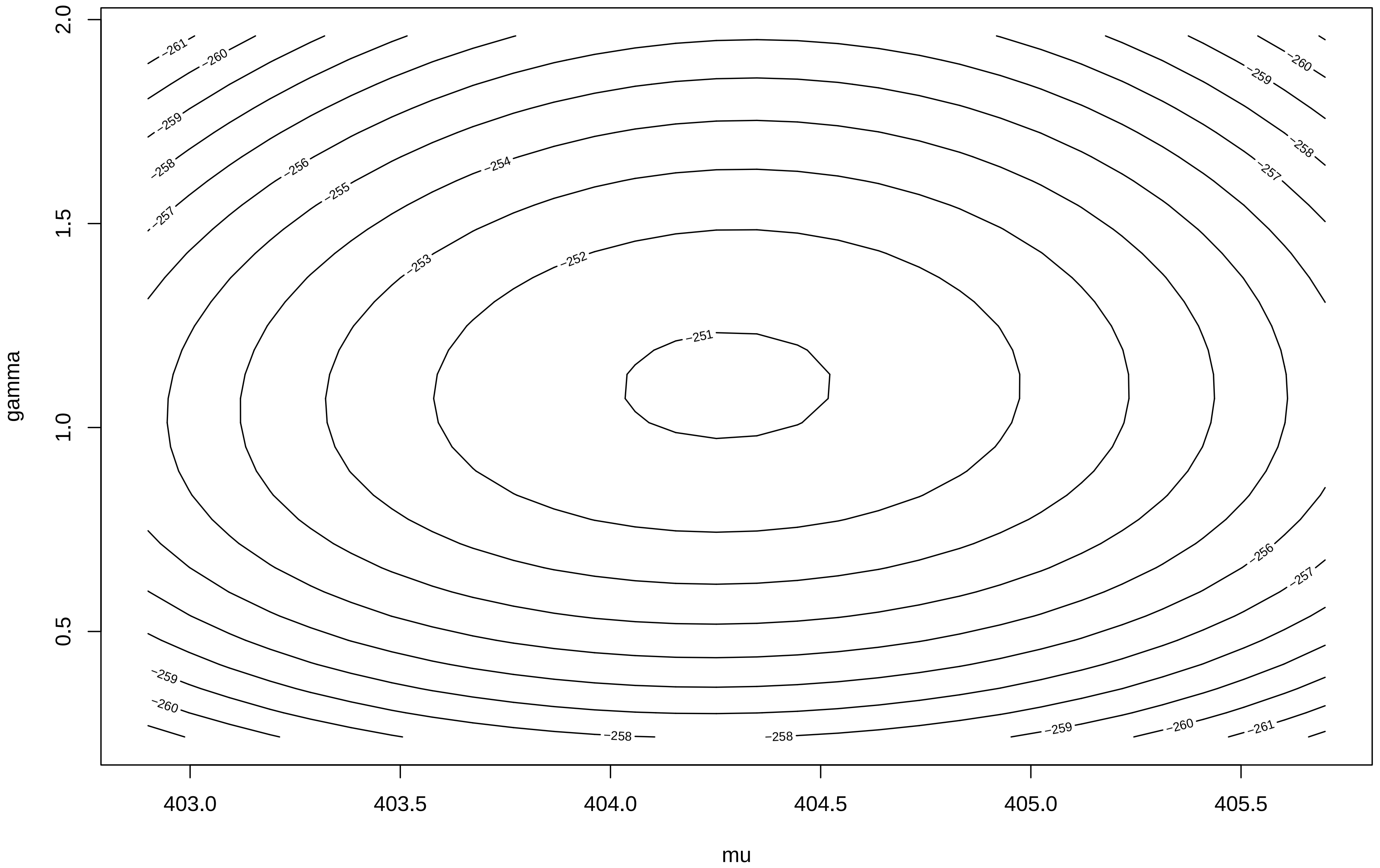
Similar code produces the **plots** for (μ, γ) and (η, γ); here's the **code** for (μ, γ):

```
mu.gamma.log.likelihood.grid <- array( data = NA, dim = c( n.grid,
  n.grid ) )

for ( i in 1:n.grid ) {
  for ( j in 1:n.grid ) {
    mu.gamma.log.likelihood.grid[ i, j ] <- log.likelihood( c( mu.grid[ i ],
      theta.hat[ 2 ], gamma.grid[ j ] ), y )
  }
}

persp( mu.grid, gamma.grid, mu.gamma.log.likelihood.grid, xlab = 'mu',
  ylab = 'gamma', zlab = 'Log Likelihood', axes = T,
  ticktype = 'detailed', theta = -30, phi = 15 )

contour( mu.grid, gamma.grid, mu.gamma.log.likelihood.grid, xlab = 'mu',
  ylab = 'gamma' )
```

The Log Likelihood Surface in (μ, γ) (3D-Perspective Plot)

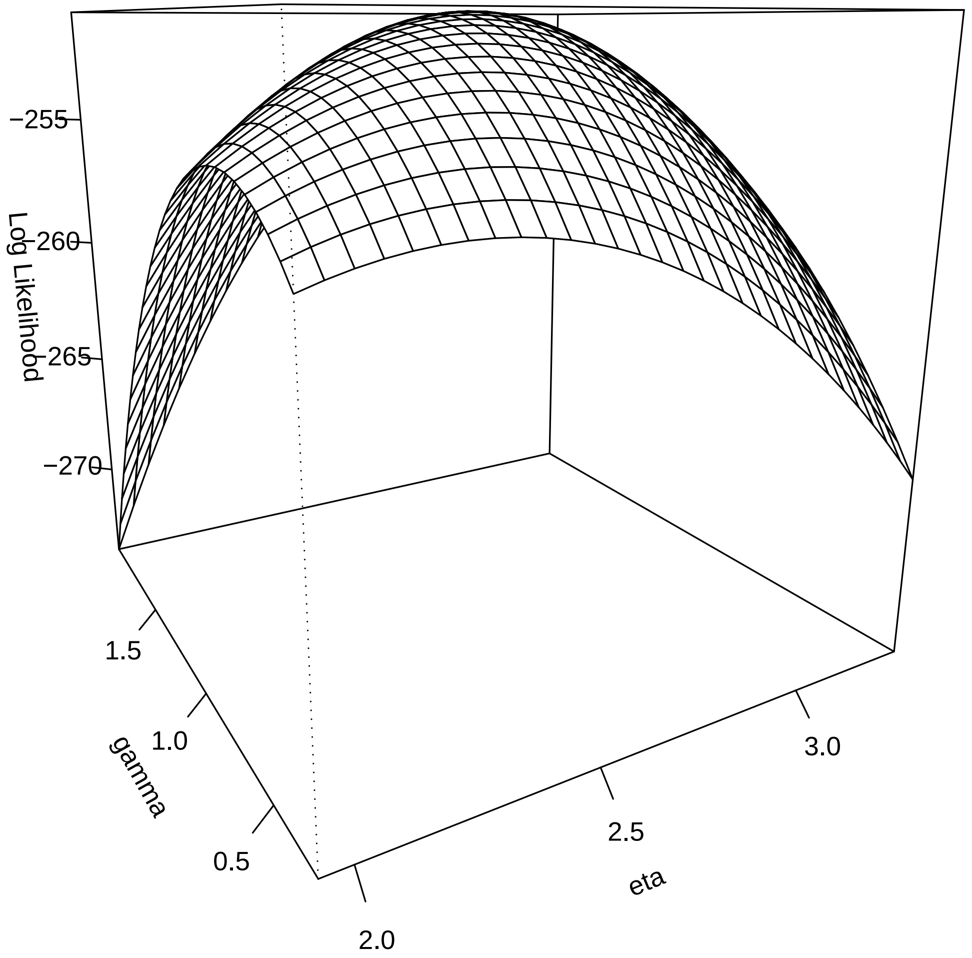


The Log Likelihood Surface in (μ, γ) (Contour Plot)

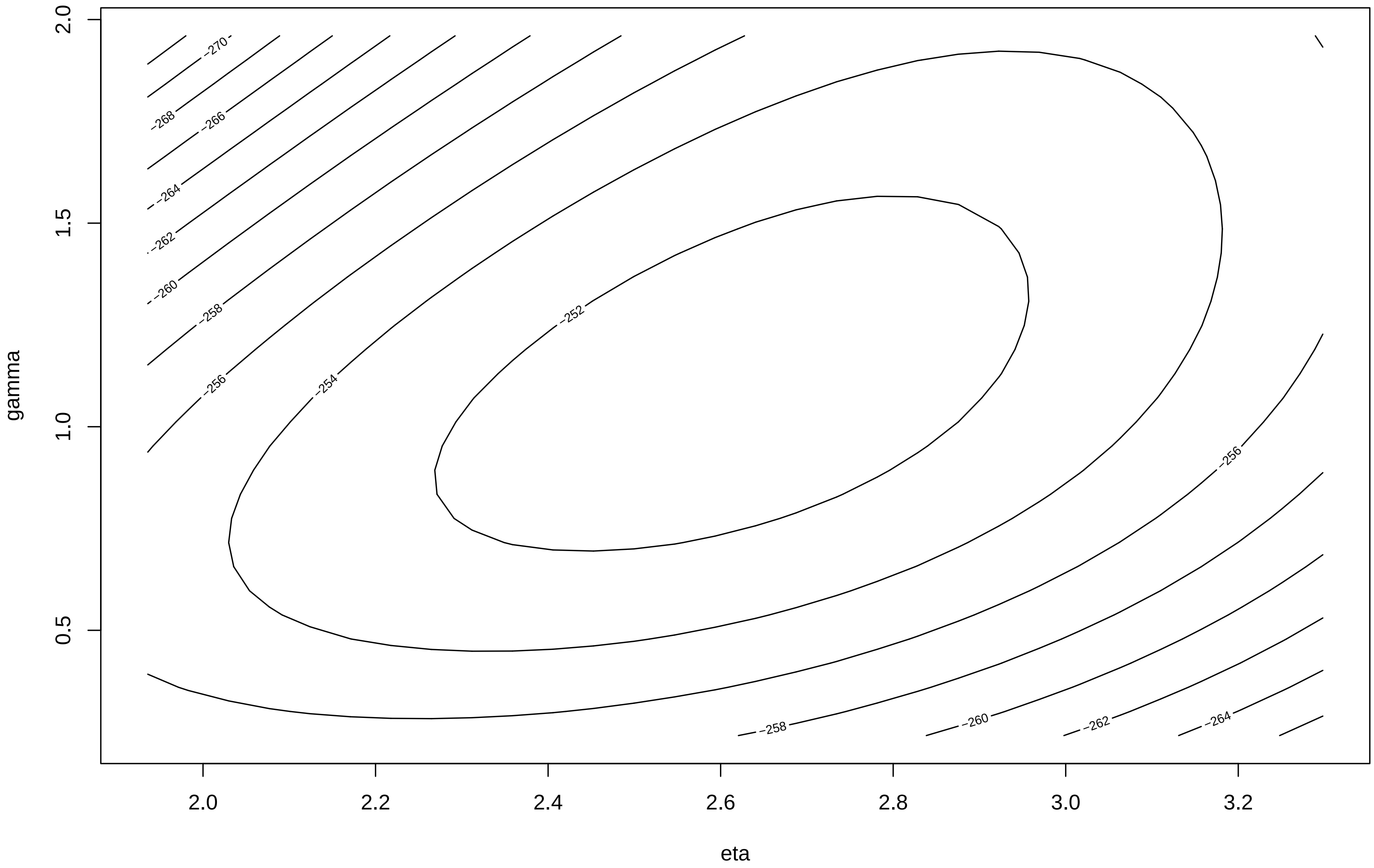


μ and γ will also be uncorrelated in the **posterior**.

The Log Likelihood Surface in (η, γ) (3D-Perspective Plot)



The Log Likelihood Surface in (η, γ) (Contour Plot)



But η and γ will be positively correlated in the **posterior**.

Ready For the MCMC Now

By the **Bernstein-von Mises Theorem** we expect the **posterior distribution** with a **diffuse prior** to be **approximately** $(\theta|y) \sim N_k\left(\hat{\theta}, \hat{\Sigma}\right)$, so we know that the **posterior SDs** of the **components** of θ are **approximately** $(\sqrt{0.2129}, \sqrt{0.05160}, \sqrt{0.08208}) \doteq (0.4614, 0.2272, 0.2865)$; this can help us **tune** the **proposal distribution SDs** (PDSDs).

If things **behave the same way** for $k = 3$ as they do for $k = 1$, we could get **acceptance rates** of about 44% using PDSDs $2.4(0.4614, 0.2272, 0.2865) \doteq 1.11, 0.545, 0.689)$; **let's see** what happens.

Here's some **quite general** R code to do **single-scan random-walk Metropolis sampling** — you can use this **driver function**, the **acceptance probability function** and the **log posterior function without change** in **any other problem**, and you'll just have to write **new code** for the **log likelihood**, **log prior** and **predictive sampling distribution** for **your specific situation**:

(see the **entry** on the **course web page** for this code and output from it)

NB10 Results

Running the **code** on the **course web page** with $M = 100{,}000$ monitoring iterations, you get the following **numerical posterior summaries**, in which I've tried not to **overstate** the **significant figures**:

```
           ---------- posterior ----------
unknown   mean   (mcse)     sd   2.5% 97.5%

     mu  404.3 (0.005) 0.446 403.4 405.2
  sigma   3.77 (0.005) 0.428  2.99  4.68
     nu    3.2 (0.012)   1.0   1.8   5.7
 y.next  404.3 (0.022)   6.9 394.9 416.6
```

With **100,000 monitoring iterations**, we've achieved **4–figure accuracy** with the **posterior mean** and **interval estimate** for μ but only **2–figure accuracy** for the **summaries** about ν (this turns out to be a **hard parameter to pin down**).

The **posterior SD** for μ, the only parameter **directly comparable** across the **Gaussian** and t models for the **NB10 data**, came out $\boxed{\mathbf{0.45}}$ from the t modeling, versus $\boxed{\mathbf{0.65}}$ with the **Gaussian**, i.e., the **interval estimate** for μ

A Model Uncertainty Anomaly?

from the **(incorrect) Gaussian model** is about **40% wider** that that from the **(much better-fitting)** t model.

$\boxed{\textbf{NB}}$ Moving from the **Gaussian** to the t model involves a **net increase** in **model uncertainty**, because when you **assume the Gaussian** you're in effect saying that you **know** the t degrees of freedom are ∞, whereas with the t model you're **treating** ν as **unknown**.

And yet, even though there's been an **increase in model uncertainty**, the **inferential uncertainty** about μ has **gone down**.

This is **relatively rare** — usually **when model uncertainty increases so does inferential uncertainty** (Draper 1995, 2011) — and arises in this case because of **two things**: (a) the t model **fits better** than the Gaussian, and (b) the **Gaussian** is actually a **conservative** model to assume as far as **inferential accuracy** for **location parameters** is concerned: it turns out that **among all symmetric unimodal densities** the **Gaussian** has **minimal Fisher information** for **location** (this is related to the **maximum-entropy property** of the **Gaussian**).