Markov chain Monte Carlo

Dr. Jarad Niemi

STAT 544 - Iowa State University

April 4, 2016

Markov chain construction

The techniques we have discussed thus far, e.g.

- Metropolis-Hastings
  - independent Metropolis-Hastings
  - random-walk Metropolis
  - Hamiltonian Monte Carlo
- Gibbs sampling
  - Slice sampling

form a set of techniques referred to as Markov chain Monte Carlo (MCMC).

Today we look at some practical questions involving the use of MCMC:

- What initial values should I use?
- How long do I need to run my chain?
- What can I do with the samples I obtain?

Markov chain Monte Carlo

An MCMC algorithm with transition kernel $K(\theta^{(t-1)}, \theta^{(t)})$ constructed to sample from $p(\theta|y)$ is the following:

1. Sample $\theta^0 \sim \pi^0$.
2. For $t = 1, \ldots, T$, perform the kernel $K(\theta^{(t-1)}, \theta^{(t)})$ to obtain a sequence $\theta^1, \ldots, \theta^{(t)}$.

The questions can then be rephrased as

- What should I use for π^0?
- What should T be?
- What can I do with $\theta^1, \ldots, \theta^{(t)}$?

Initial values

For ergodic Markov chains with stationary distribution $p(\theta|y)$, theory states that

$$\theta^{(t)} \xrightarrow{d} \theta \text{ where } \theta \sim p(\theta|y)$$

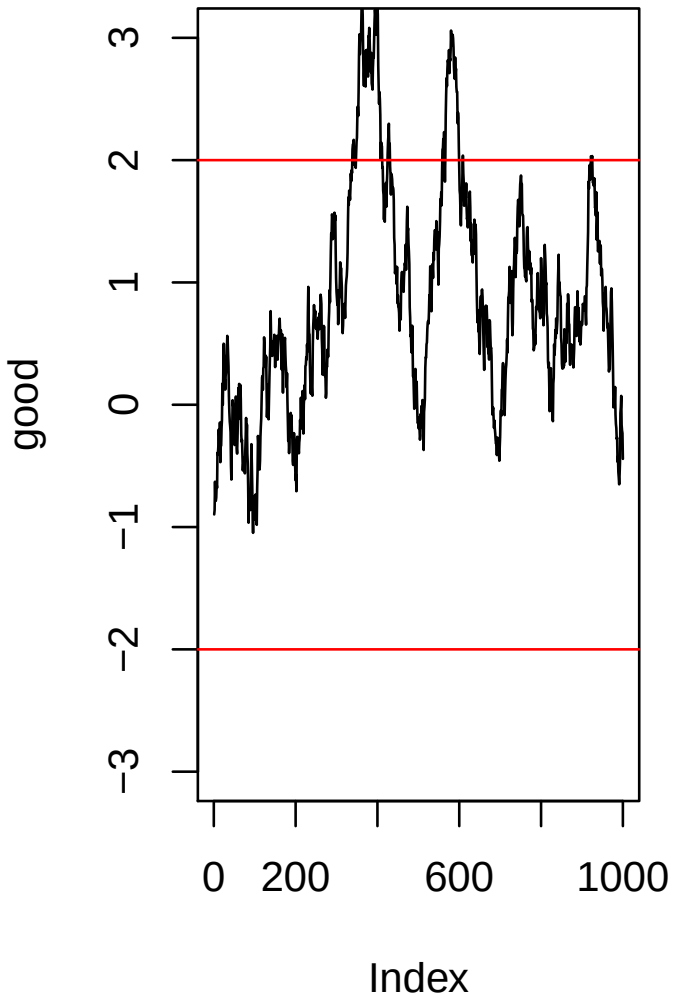
for almost all θ^0 (all with Harris recurrence).

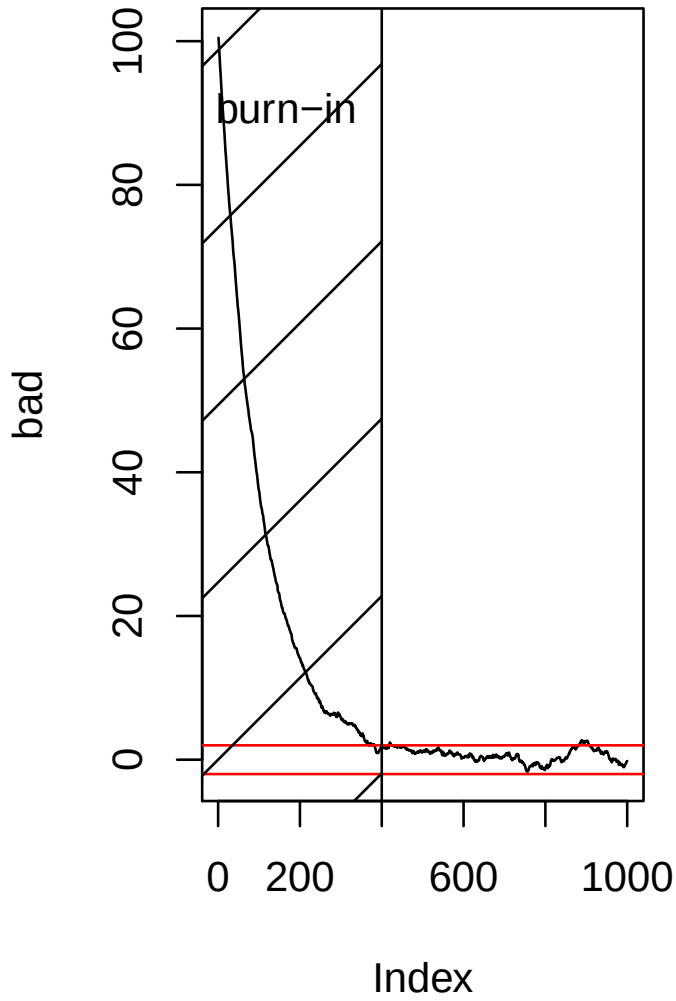
If $p(\theta^0|y) << p(\theta_{MAP}|y)$, then this can take a long time. For example, let

$$\theta^{(t)} = 0.99\theta^{(t-1)} + \epsilon_t \qquad \epsilon_t \overset{iid}{\sim} N(0, 1 - .99^2)$$

which has stationary distribution $p(\theta|y) \overset{d}{=} N(0,1)$. If

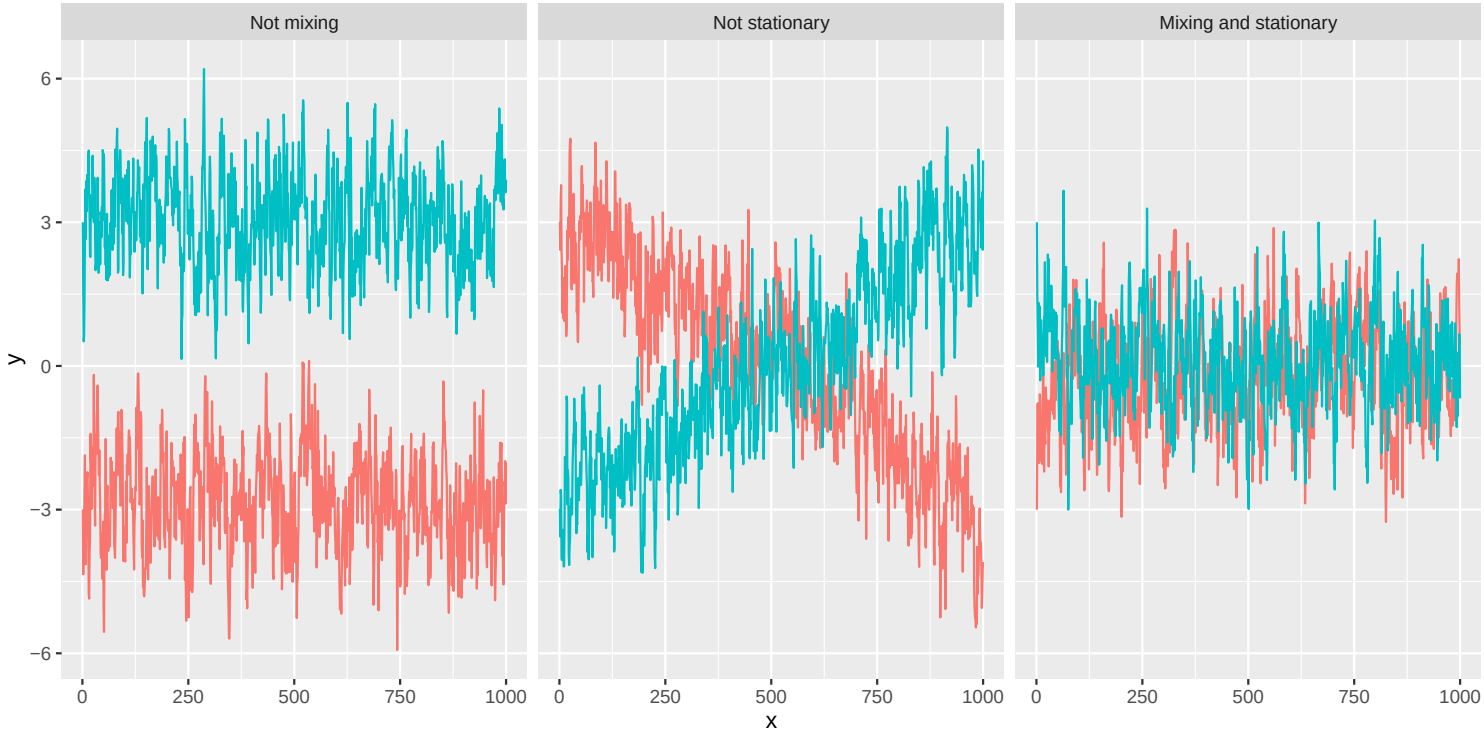
- $\theta^0 \sim p(\theta|y)$ then $\theta^{(t)} \dot{\sim} p(\theta|y)$ for all t, but if
- θ^0 is very far from $p(\theta|y)$ then $\theta^{(t)} \dot{\sim} p(\theta|y)$ only for t very large.


good
3
2
1
0
−1
−2
−3
0
200
600
1000
Index


bad
100
80
60
40
20
0
burn−in
0
200
600
1000
Index

How many iterations do I need for burn-in?

Imagine two different chains



Gelman-Rubin potential scale reduction factor

1. Start multiple chains with initial values that are well dispersed values relative to $p(\theta|y)$.
2. For each scalar estimand ψ of interest,
   - Calculate the between B and within W chain variances
   - Estimate the the marginal posterior variance of the estimand, i.e. $Var(\psi|y)$:

     $$\widehat{Var}^{+}(\psi|y) = \frac{t-1}{t}W + \frac{1}{t}B$$

     where t is the number of iterations.
   - Calculate the potential scale reduction factor

     $$\hat{R}_{\psi} = \sqrt{\frac{\widehat{Var}^{+}(\psi|y)}{W}}$$

3. If the $\hat{R}_{\psi}$ are approximately 1, e.g. <1.1, then there is no evidence of non-convergence.

Example potential scale reduction factors

```
[1] "Not mixing"
Potential scale reduction factors:

     Point est. Upper C.I.
[1,]       7.35       16.2

[1] "Not stationary"
Potential scale reduction factors:

     Point est. Upper C.I.
[1,]       2.62       5.31

[1] "Mixing and stationary"
Potential scale reduction factors:

     Point est. Upper C.I.
[1,]       1.01       1.04
```

Methods for finding good initial values

From `http://users.stat.umn.edu/~geyer/mcmc/burn.html`:

> *Any point you don't mind having in a sample is a good starting point.*

Methods for finding good initial values:

- burn-in: throw away the first X iterations
- Start at the MLE, i.e. $\text{argmax}_\theta p(y|\theta)$
- Start at the MAP (maximum aposterior), i.e. $\text{argmax}_\theta p(\theta|y)$

How many iterations should I run (post 'convergence')?

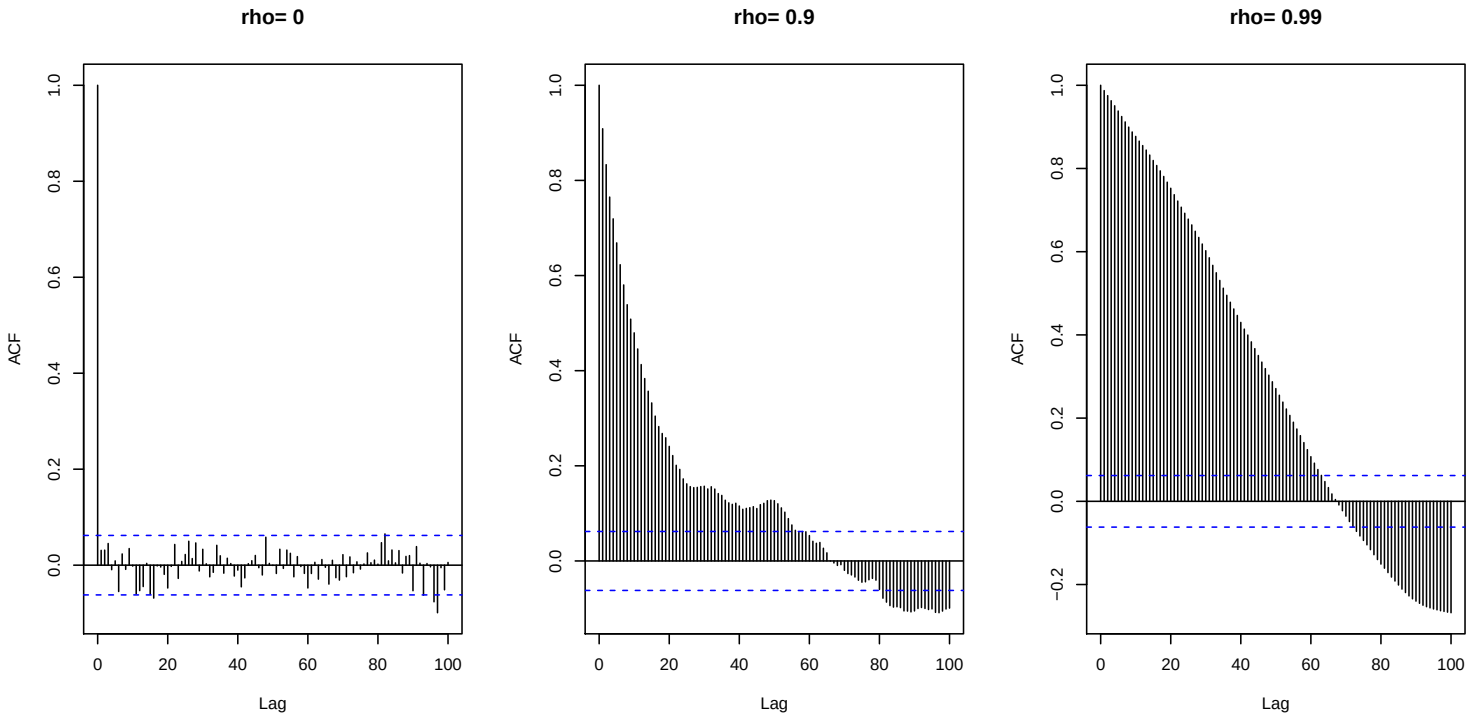
Compute the effective sample size, i.e. how many independent samples would we need to get the equivalent precision of our estimates?

```
d = ddply(data.frame(rho=c(0,.9,.99)), .(rho), function(x) data.frame(x=rwalk(1000,0,x$rho)))
ddply(d, .(rho), summarize, effective_size = coda::effectiveSize(x))
```

```
   rho effective_size
1 0.00    1000.000000
2 0.90      35.405683
3 0.99       6.435595
```

BDA3 a total of 100-2000 effective samples. But this really depends on what you want to estimate. If you are interested in estimating probabilities of rare events, i.e. tail probabilities, you may need many more samples.

Autocorrelation function



Monte Carlo integration

Consider approximating the integral via it's Markov chain Monte Carlo (MCMC) estimate, i.e.

$$E_{\theta|y}[h(\theta)|y] = \int_{\Theta} h(\theta)p(\theta|y)d\theta \quad \text{and} \quad \hat{h}_T = \frac{1}{T}\sum_{t=1}^{(t)} h\left(\theta^{(t)}\right).$$

where $\theta^{(t)}$ is the t^{th} iteration from the MCMC. Under regularity conditions,

- SLLN: $\hat{h}_T$ converges almost surely to $E[h(\theta)|y]$.
- CLT: under stronger regularity conditions, then

$$\hat{h}_T \xrightarrow{d} N(E[h(\theta)|y], \sigma^2/T)$$

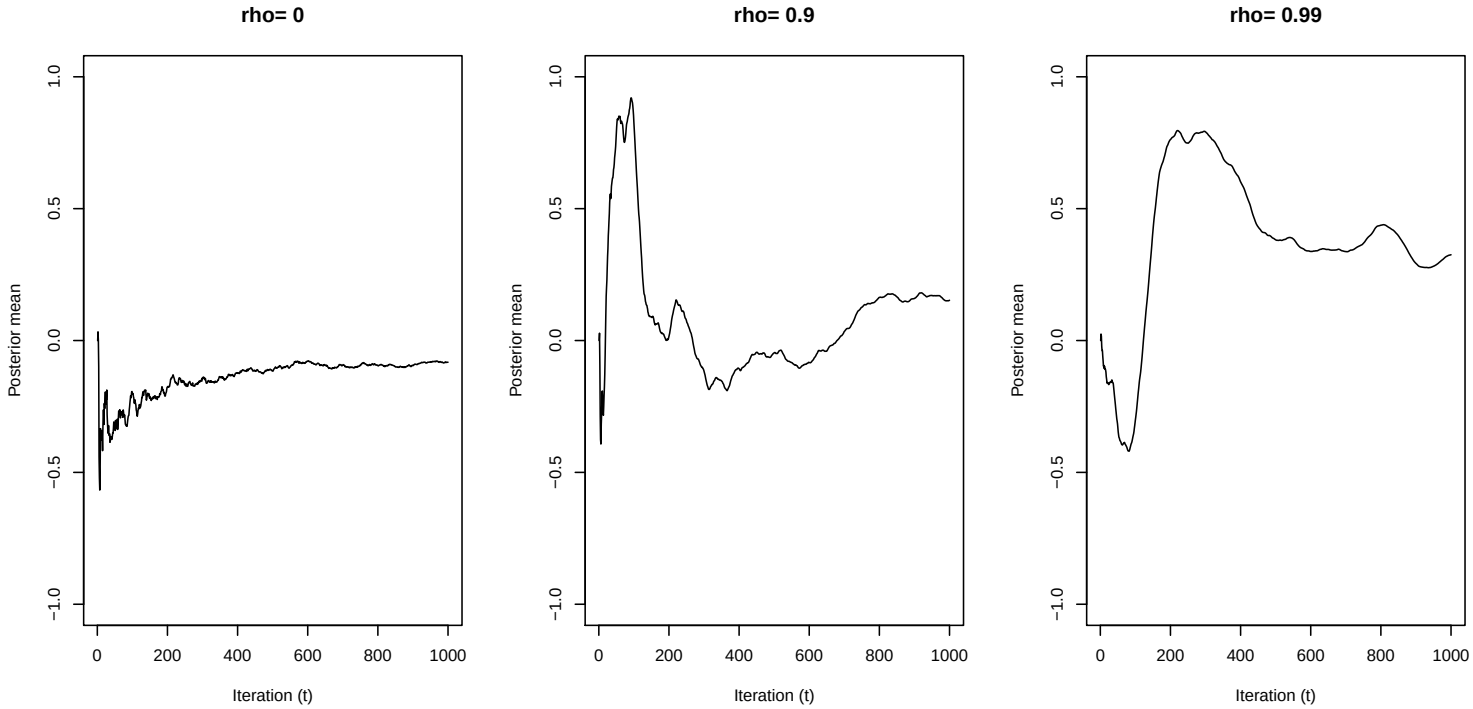
where

$$\sigma^2 = Var[h(\theta)|y]\left(1 + 2\sum_{k=1}^{\infty}\rho_k\right)$$

where ρ_k is the k^{th} autocorrelation of the $h(\theta)$ values.

Sequential estimates

```
opar = par(mfrow=c(1,3))
d_ply(d, .(rho), function(x)
  plot(cumsum(x$x)/1:length(x$x), type="l", ylim=c(-1,1),
       ylab="Posterior mean", xlab="Iteration (t)", main=paste("rho=", x$rho[1]))
)
```




```
par(opar)
```

Treat the MCMC samples as samples from the posterior

Use `mcmcse::mcse` to estimate the MCMC variance

```
# Mean
ddply(d, .(rho), function(x) as.data.frame(mcmcse::mcse(x$x)))
```

```
   rho         est         se
1 0.00 -0.08223773 0.02789422
2 0.90  0.15262785 0.13563890
3 0.99  0.32514041 0.15349268
```

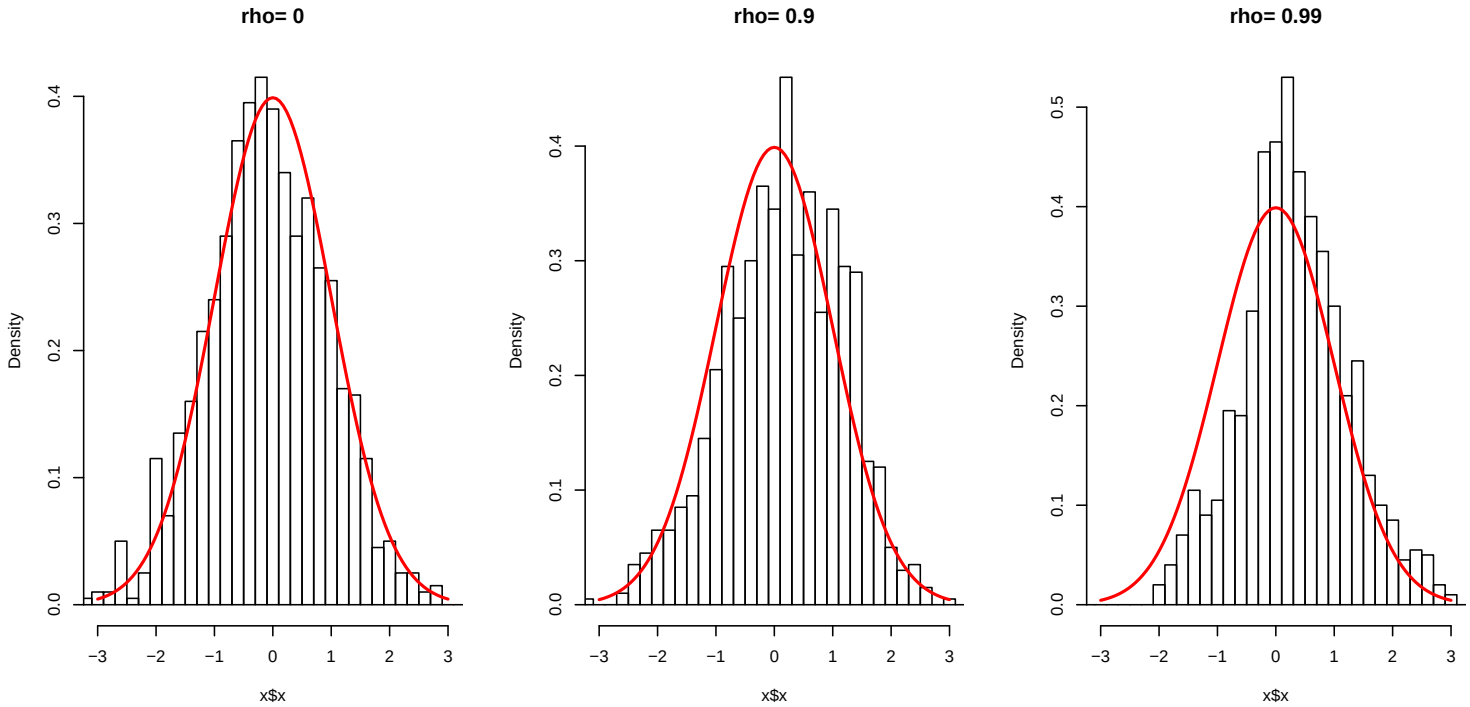
```
# Quantiles
ddply(d, .(rho), function(x) as.data.frame(mcmcse::mcse(x$x< qnorm(0.025))))
```

```
   rho   est          se
1 0.00 0.038 0.006307577
2 0.90 0.030 0.017307791
3 0.99 0.002 0.002008050
```

```
ddply(d, .(rho), function(x) as.data.frame(mcmcse::mcse(x$x< qnorm(0.975))))
```

```
   rho   est          se
1 0.00 0.976 0.005587368
2 0.90 0.976 0.012659026
3 0.99 0.954 0.027747172
```

Treat the MCMC samples as samples from the posterior



A wasteful approach

The Gelman approach in practice is the following

1. Run an initial chain or, in some other way, approximate the posterior.
2. (Randomly) choose initial values for multiple chains well dispersed relative to this approximation to the posterior.
3. Run the chain until all estimands of interest have potential scale reduction factors less than 1.1.
4. Continuing running until you have a total of around 4,000 effective draws.
5. Discard the first half of all the chains.

Assuming this approach correctly diagnosis convergence or lack thereof, it seems computationally wasteful since

- You had to run an initial chain, but then threw it away.
- You threw away half of your later iterations.

One really long chain

From `http://users.stat.umn.edu/~geyer/mcmc/one.html`

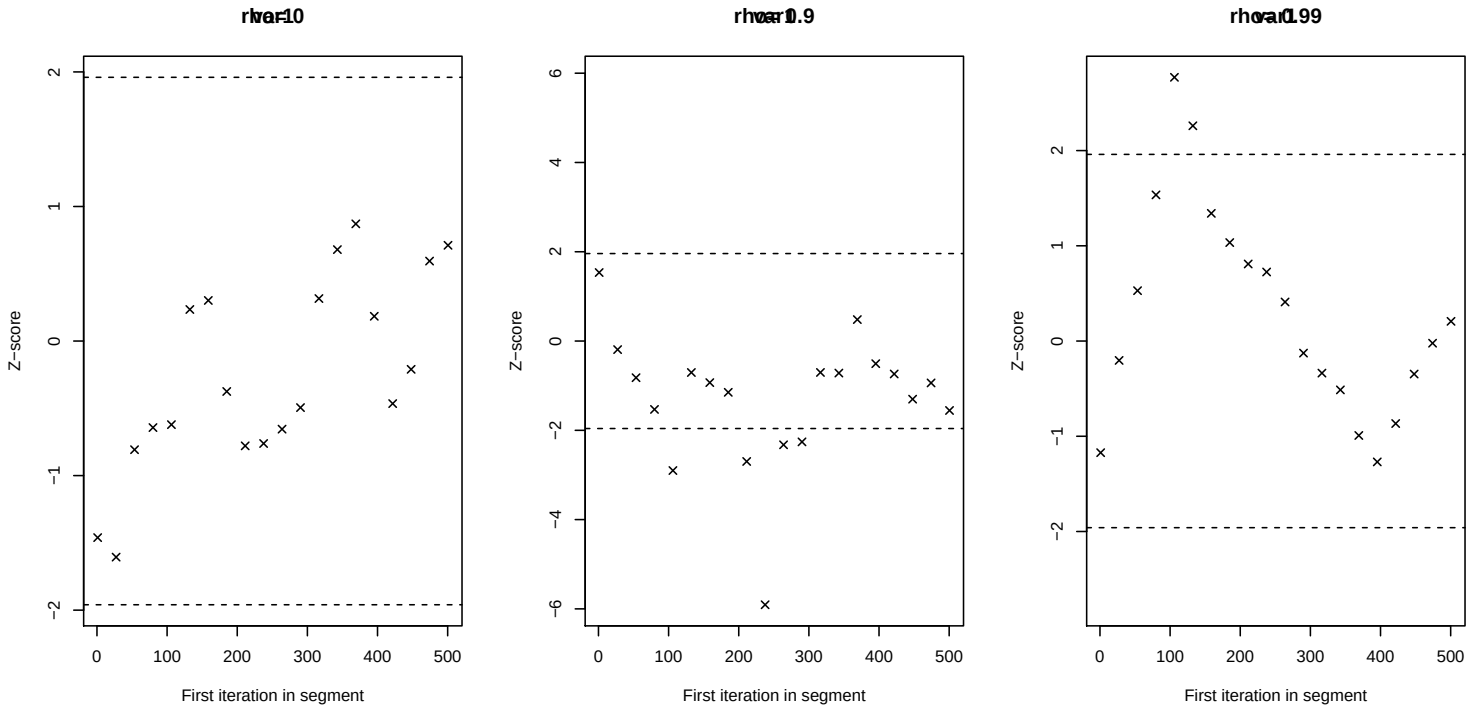
> *If you can't get a good answer with one long run, then you can't get a good answer with many short runs either.*

1. Start a chain at a reasonable starting value.
2. Run it for many iterations (and keep running it).

If you really want a convergence diagnostic, you can try Geweke's which tests for equality of means in the first and last parts of the chain.

Geweke diagnostic

```
# Z-score for test of equality of means
par(mfrow=c(1,3))
d_ply(d, .(rho), function(x) geweke.plot(mcmc(x$x), auto=F, main=paste("rho=", x$rho[1])))
```




```
par(opar)
```

Thinning

You will hear of people thinning their Markov chain by only recording every n^{th} observation.

This has the benefit of reducing the autocorrelation in the retained samples.

But should only be used if memory or hard drive space is a limiting factor.

Thinning

```
sq = seq(10,1000,by=10)
ddply(d, .(rho), summarize, full=effectiveSize(x), thinned=effectiveSize(x[sq]))
```

```
   rho        full   thinned
1 0.00 1000.000000 103.29644
2 0.90   35.405683  39.37303
3 0.99    6.435595  16.21098
```

```
# Calculate standard error
ddply(d, .(rho), function(x) {
  rbind(data.frame(s="full", mcmcse::mcse(x$x)),data.frame(s="thinned", mcmcse::mcse(x$x[sq])))
})
```

```
   rho       s         est         se
1 0.00    full -0.08223773 0.02789422
2 0.00 thinned -0.16062926 0.08828254
3 0.90    full  0.15262785 0.13563890
4 0.90 thinned  0.19911506 0.16895797
5 0.99    full  0.32514041 0.15349268
6 0.99 thinned  0.32068486 0.26609394
```

Alternative use for burn-in

For MCMC algorithms that have tuning parameters, use burn-in (warm-up) to tune tuning parameters.

Suppose the target distribution is $N(0, 1)$ and we are performing a random-walk Metropolis with a normal proposal. The variance of this proposal is a tuning parameter and we can tune it during burn-in:

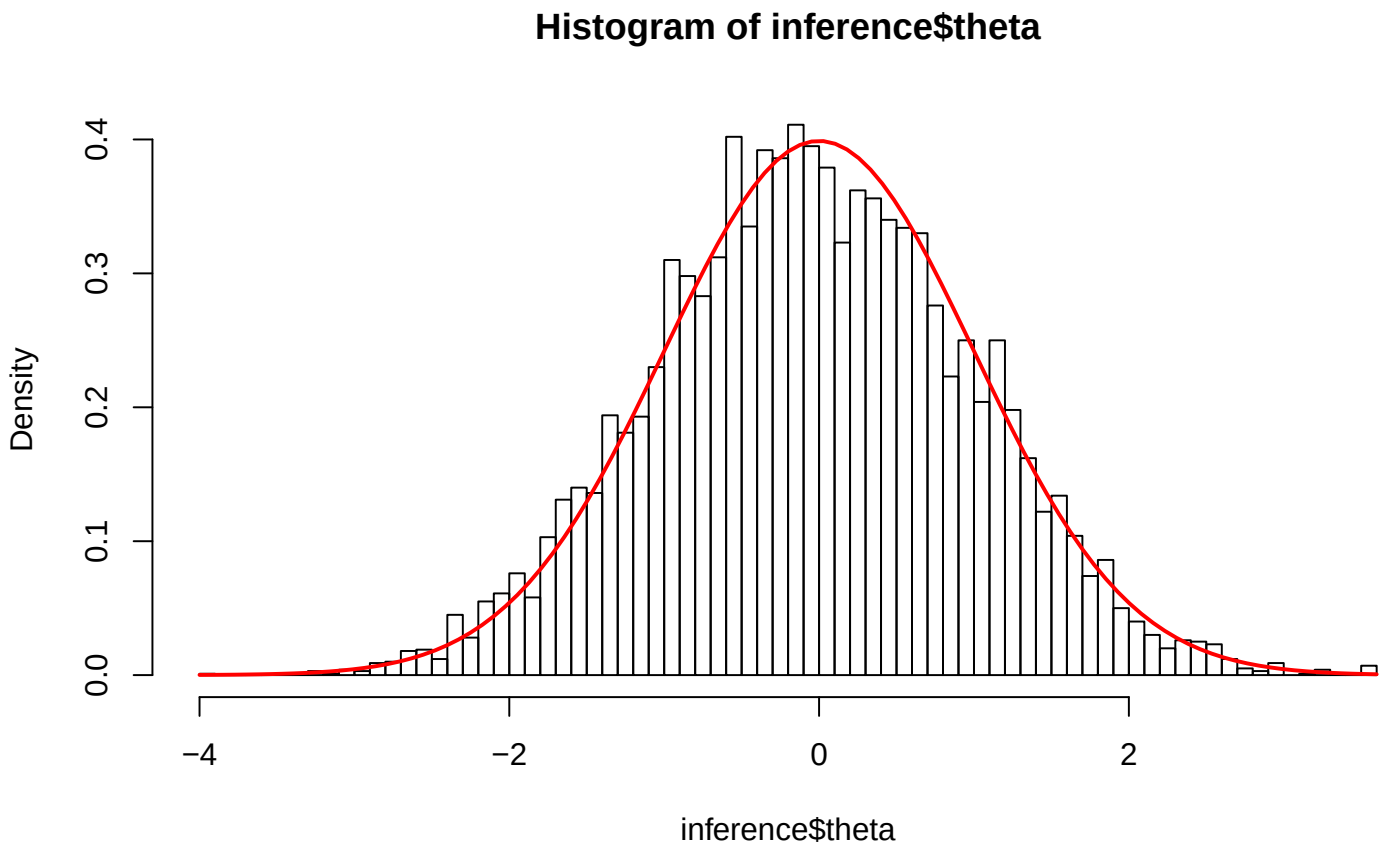
- if a proposal is accepted, then likely our variance is too small and therefore we should increase it
- if a proposal is rejected, then likely our variance is too big and therefore we should decrease it

Alternative use for burn-in

```
rw = function(n, theta0, tune=1, autotune=TRUE) {
  theta = rep(theta0, n)
  for (i in 2:n) {
    theta_prop = rnorm(1, theta[i-1], tune)
    logr = dnorm(theta_prop, log=TRUE) - dnorm(theta[i-1], log=TRUE)

    # This tuning tunes to an acceptance rate of 50%
    if (log(runif(1))<logr) {
      theta[i] = theta_prop
      if (autotune) tune = tune*1.1
    } else {
      theta[i] = theta[i-1]
      if (autotune) tune = tune/1.1
    }
  }
  return(list(theta=theta,tune=tune))
}

# Tune during burn-in
burnin = rw(1000, 0)
burnin$tune


[1] 1.61051


# Turn off tuning after burn-in for theory to hold
inference = rw(10000, burnin$theta[1000], burnin$tune, autotune=FALSE)
```

Alternative use for burn-in

```
hist(inference$theta, 100, prob=T)
curve(dnorm, col="red", add=TRUE, lwd=2)
```



Summary

Since computing time/power is not very limited these days, my suggestion is

1. Run one long chain and continue running it
2. Run multiple chains according to suggestions in BDA
    a. Start multiple chains with initial values relative to the posterior learned by the long chain
    b. Monitor the potential scale reduction factor until < 1.1 for all quantities of interest
    c. Monitor traceplots and cumulative mean plots
    d. Discard burn-in (first half is probably overkill)
    e. Run until effective sample size is around 2000
3. Use all samples for posterior inference

If things are not going well,

1. Check for identifiability of the parameters in your model.
2. Construct a better sampler.

A simple model

Let

$$Y_i \overset{ind}{\sim} N(\mu, \sigma^2) \qquad \text{and} \qquad p(\mu, \sigma) \propto Ca^+(\sigma; 0, 1)$$

In RStan,

```
model = "
data {
  int<lower=1> n;
  real y[n];
}
parameters{
  real mu;
  real<lower=0> sigma;
}
model {
  sigma ~ cauchy(0,1);
  y ~ normal(mu,sigma);
}
"
```

RStan

```
y = rnorm(10)
m = stan_model(model_code = model)
r = sampling(m, list(n=length(y), y=y))
```

```
r
```

```
Inference for Stan model: 6c86a547f723283854dd490525d54ee4.
4 chains, each with iter=2000; warmup=1000; thin=1;
post-warmup draws per chain=1000, total post-warmup draws=4000.

       mean se_mean   sd  2.5%   25%   50%   75% 97.5% n_eff Rhat
mu    -0.30    0.01 0.38 -1.07 -0.55 -0.31 -0.07  0.47  1323    1
sigma  1.16    0.01 0.30  0.75  0.94  1.10  1.30  1.89  1381    1
lp__  -6.90    0.03 1.12 -9.77 -7.31 -6.55 -6.11 -5.83  1032    1

Samples were drawn using NUTS(diag_e) at Mon Apr  4 15:18:31 2016.
For each parameter, n_eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor on split chains (at
convergence, Rhat=1).
```

```
laply(extract(r, c("mu","sigma")), function(x) length(unique(x))/length(x)) # Acceptance rate
```

```
[1] 0.95975 0.95975
```

RStan plot