Additional notes on MCMC sampling

Shravan Vasishth

March 16, 2020

For more details on MCMC, some good books are:

1. The MCMC handbook by Brooks et al. [1]
2. Gilks et al [2]
3. Lynch [3]

1 Introduction: Monte Carlo integration

Monte Carlo integration sounds fancy, but basically this amounts to sampling from a distribution f(x), and computing summaries like the mean. Formally, we calculate E(f(X)) by drawing samples $\{X_1, \ldots, X_n\}$ and then approximating:

$$E(f(X)) \approx \frac{1}{n} \sum f(X) \tag{1}$$

For example:

```
x<-rnorm(1000,mean=0,sd=1)
mean(x)

## [1] 0.014347
```

We can increase sample size to as large as we want.
We can also compute quantities like $P(X < 1.96)$ by sampling:

```r
mean((x<1.96))
```

```
## [1] 0.977
```

```r
## theoretical value:
pnorm(1.96)
```

```
## [1] 0.975
```

So, we can compute summary statistics using simulation. However, if we only know up to proportionality the form of the distribution to sample from, how do we get these samples to summarize from? Monte Carlo Markov Chain (MCMC) methods provide that capability: they allow you to sample from distributions you only know up to proportionality.

1.1 Markov chain sampling

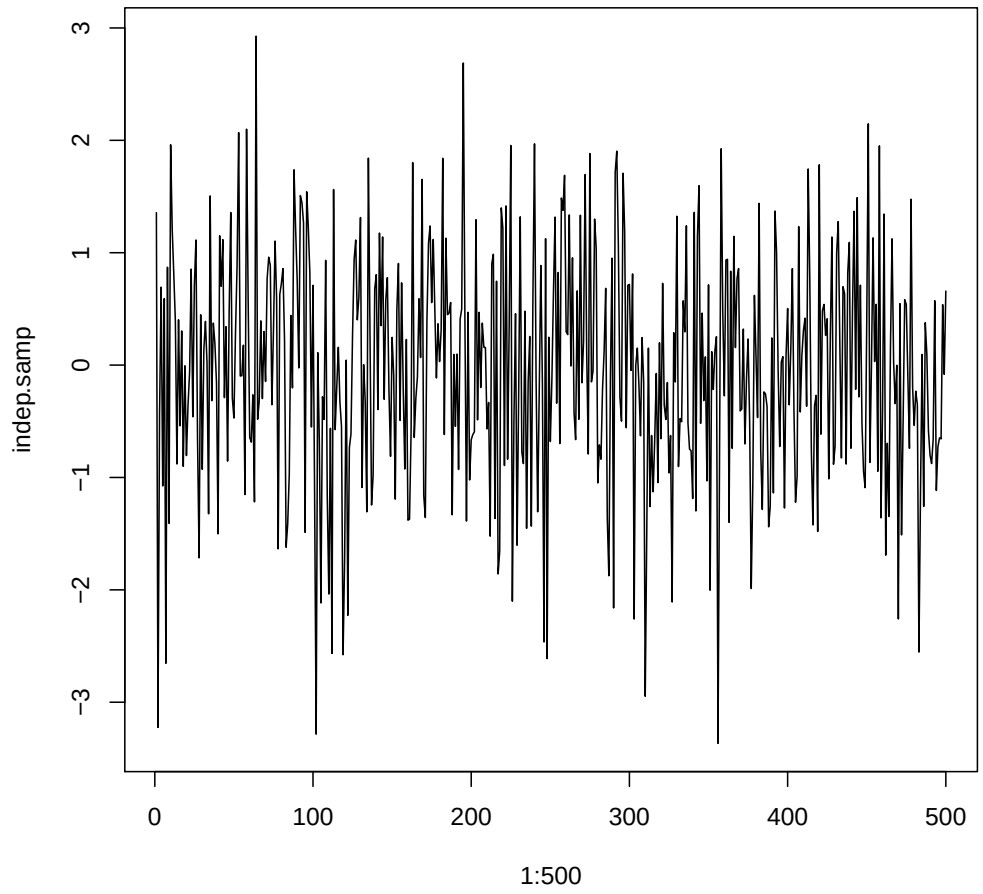
We have been doing non-Markov chain sampling when we started this course. Taking independent samples is an example of non-Markov chain sampling:

```r
indep.samp<-rnorm(500,mean=0,sd=1)
head(indep.samp)
```

```
## [1]  1.35654 -3.22375 -0.43777  0.69353 -1.07536  0.59214
```

The vector of values sampled here are statistically independent.

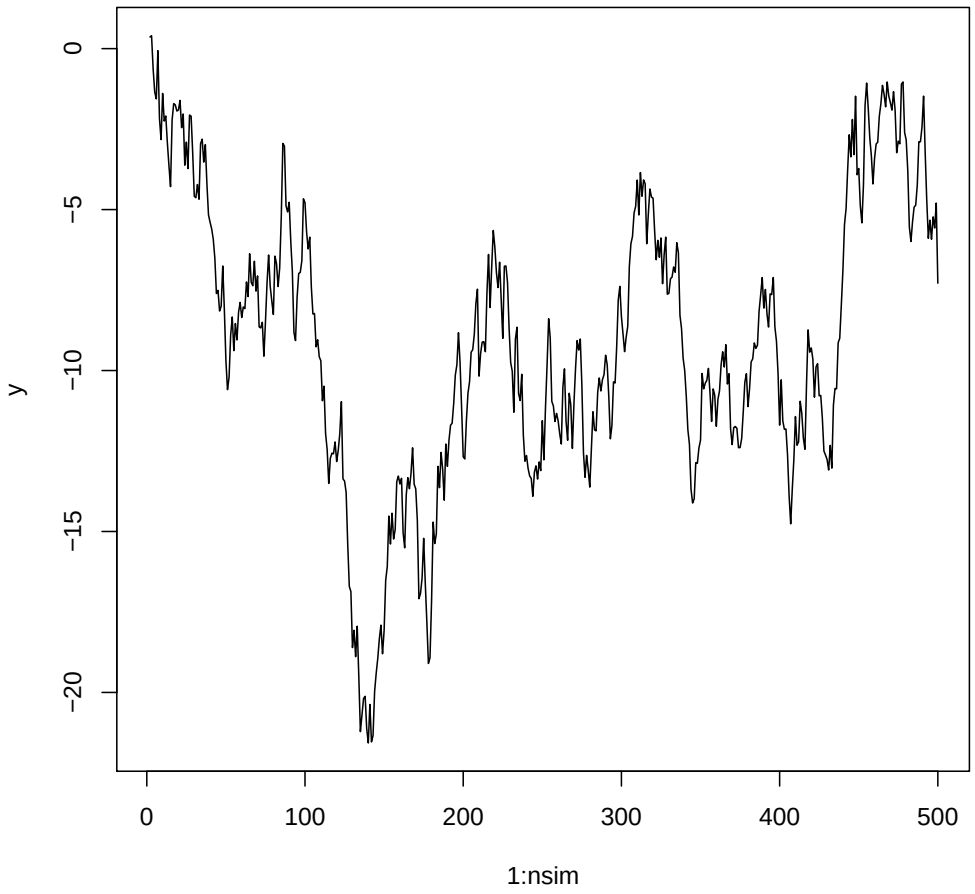
```r
plot(1:500,indep.samp,type="l")
```



If the current value influences the next one, we have a Markov chain. Here is a simple Markov chain: the i-th draw is dependent on the i-1 th draw:

```
nsim<-500
x<-rep(NA,nsim)
y<-rep(NA,nsim)
x[1]<-rnorm(1) ## initialize x
for(i in 2:nsim){
## draw i-th value based on i-1-th value:
y[i]<-rnorm(1,mean=x[i-1],sd=1)
x[i]<-y[i]
```

```
}
plot(1:nsim,y,type="l")
```



1.1.1 What is convergence in a Markov chain? An illustration using discrete Markov chains

A discrete Markov chain defines probabilistic movement from one state to the next. Suppose we have 6 states; a **transition matrix** can define the probabilities:

```
## Set up transition matrix:
T<-matrix(rep(0,36),nrow=6)
diag(T)<-0.5
offdiags<-c(rep(0.25,4),0.5)
for(i in 2:6){
T[i,i-1]<-offdiags[i-1]
}
offdiags2<-c(0.5,rep(0.25,4))
for(i in 1:5){
T[i,i+1]<-offdiags2[i]
}
T
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,] 0.50 0.50 0.00 0.00 0.00 0.00
## [2,] 0.25 0.50 0.25 0.00 0.00 0.00
## [3,] 0.00 0.25 0.50 0.25 0.00 0.00
## [4,] 0.00 0.00 0.25 0.50 0.25 0.00
## [5,] 0.00 0.00 0.00 0.25 0.50 0.25
## [6,] 0.00 0.00 0.00 0.00 0.50 0.50
```

Note that the rows sum to 1, i.e., the probability mass is distributed over all possible transitions from any one location:

```
rowSums(T)
```

```
## [1] 1 1 1 1 1 1
```

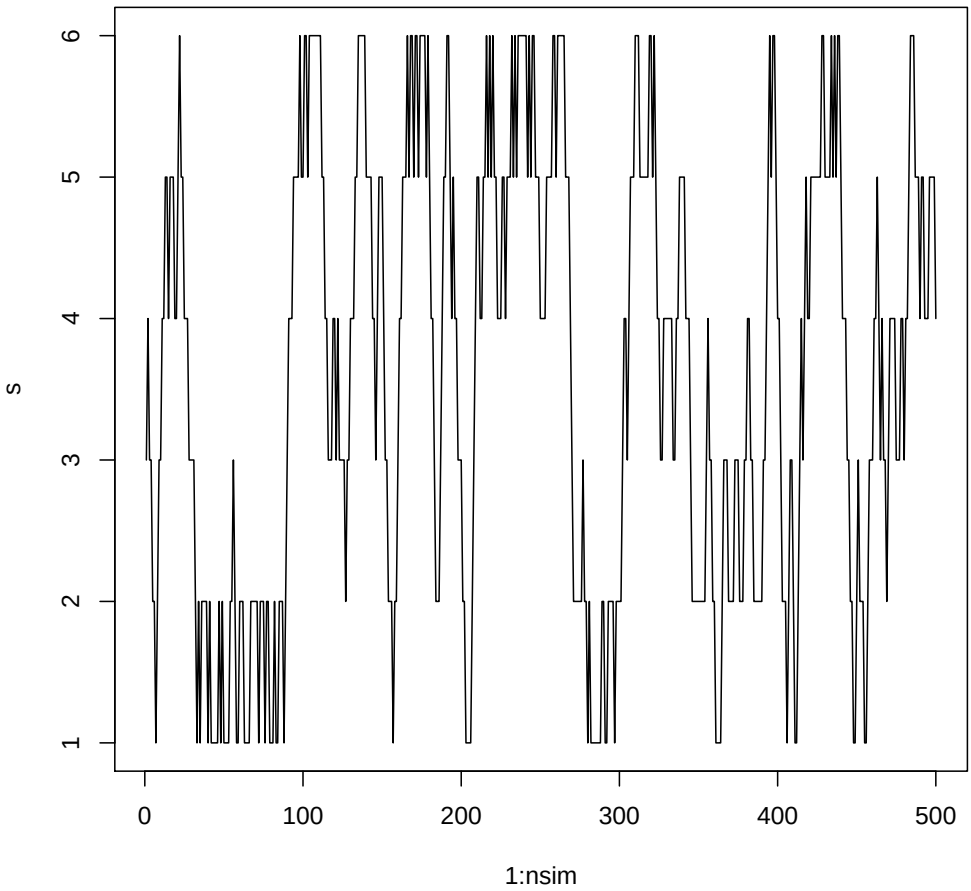
We can represent a current location as a probability vector: e.g., in state one, the transition probabilities for possible moves are:

```
T[1,]
```

```
## [1] 0.5 0.5 0.0 0.0 0.0 0.0
```

We can also simulate a random walk based on T:

```r
nsim<-500
s<-rep(0,nsim)
## initialize:
s[1]<-3
for(i in 2:nsim){
  s[i]<-sample(1:6,size=1,prob=T[s[i-1],])
}

plot(1:nsim,s,type="l")
```

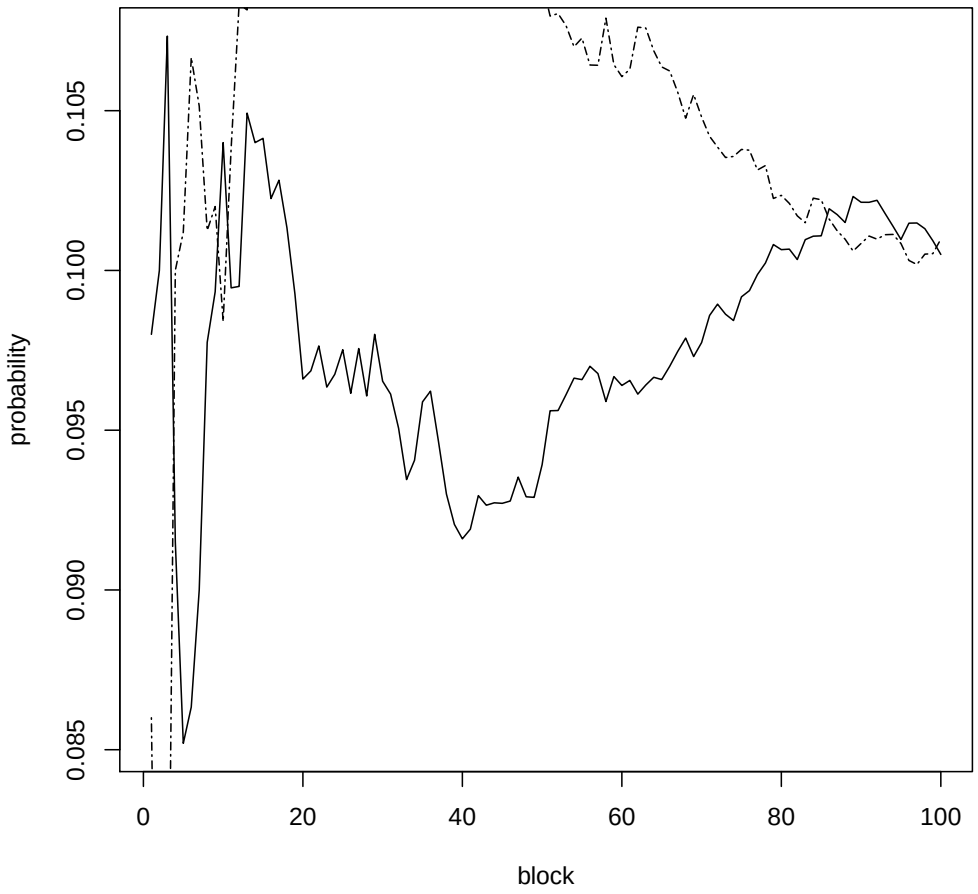


Note that the above random walk is non-deterministic: if we rerun the

above code multiple times, we will get different patterns of movement.

This Markov chain converges to a particular distribution of probabilities of visiting states 1 to 6. We can see the convergence happen by examining the proportions of visits to each state after blocks of steps that increase by 500 steps:

```
nsim<-50000
s<-rep(0,nsim)
## initialize:
s[1]<-3
for(i in 2:nsim){
  s[i]<-sample(1:6,size=1,prob=T[s[i-1],])
}


blocks<-seq(500,50000,by=500)
n<-length(blocks)
## store transition probs over increasing blocks:
store.probs<-matrix(rep(rep(0,6),n),ncol=6)
## compute relative frequencies over increasing blocks:
for(i in 1:n){
  store.probs[i,]<-table(s[1:blocks[i]])/blocks[i]
}

## convergence for state 1:
for(j in 1:6){
if(j==1){
plot(1:n,store.probs[,j],type="l",lty=j,xlab="block",
     ylab="probability")
} else {
  lines(1:n,store.probs[,j],type="l",lty=j)
}
}
```



Note that each of the rows of the store.probs matrix is a probability mass function, which defines the probability distribution for the 6 states:

```
store.probs[1,]

## [1] 0.098 0.172 0.178 0.208 0.258 0.086
```

This distribution is settling down to a particular set of values; that's what we mean by convergence. This particular set of values is:

```
(w<-store.probs[n,])

## [1] 0.10050 0.20564 0.20420 0.19498 0.19370 0.10098
```

w is called a **stationary** distribution. If $wT = w$, then then w is the stationary distribution of the Markov chain.

```
round(w%*%T,digits=2)

##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]  0.1  0.2  0.2  0.2  0.2  0.1

round(w,digits=2)

## [1] 0.10 0.21 0.20 0.19 0.19 0.10
```

This discrete example gives us an intuition for what will happen in continuous distributions: we will devise a Markov chain such that the chain will converge to the distribution we are interested in sampling from.

2 Monte Carlo sampling

Suppose we have a posterior that has the form of a Beta distribution. We can sample from the posterior distribution easily:

```
x<-rbeta(5000,1498,1519)
```

Once we have these samples, we can compute any kind of useful summary, e.g., the posterior probability p (given the data) that $p > 0.5$:

```
table(x>0.5)[2]/ sum(table(x>0.5))

##  TRUE
## 0.352
```

Or we can compute an interval over which we are 95% sure that the true parameter value lies:

```r
##lower bound:
quantile(x,0.025)

##     2.5%
## 0.47818

## upper bound:
quantile(x,0.975)

##    97.5%
## 0.51443
```

Since we can integrate the Beta distribution analytically, we could have done the same thing with the `qbeta` function (or simply using calculus):

```r
(lower<-qbeta(0.025,shape1=1498,shape2=1519))

## [1] 0.47869

(upper<-qbeta(0.975,shape1=1498,shape2=1519))

## [1] 0.51436
```

Using calculus (well, we are still using R; I just mean that one could do this by hand, by solving the integral):

```r
integrate(function(x) dbeta(x,shape1=1498,shape2=1519),
          lower=lower,upper=upper)

## 0.95 with absolute error < 9.9e-12
```

However—and here we finally get to the crucial point—integration of posterior densities is often impossible (e.g., because they may have many dimensions). In those situations we use sampling methods called Markov chain Monte Carlo, or MCMC, methods.

First, let's look at two relatively simple methods of sampling.

3 The inversion method

This method works when we can know the closed form of the pdf we want to simulate from and can derive the inverse of that function.

Steps:

1. Sample one number u from $Unif(0,1)$. Let $u = F(z) = \int_L^z f(x)\,dx$ (here, L is the lower bound of the pdf f).
2. Then $z = F^{-1}(u)$ is a draw from $f(x)$.

Example: let $f(x) = \frac{1}{40}(2x+3)$, with $0 < x < 5$. We have to draw a number from the uniform distribution and then solve for z, which amounts to finding the inverse function:

$$u = \int_0^z \frac{1}{40}(2x+3) \tag{2}$$

```
u<-runif(1000,min=0,max=1)

z<-(1/2) * (-3 + sqrt(160*u +9))
```

This method can't be used if we can't find the inverse, and it can't be used with multivariate distributions.

4 The rejection method

If $F^{-1}(u)$ can't be computed, we sample from $f(x)$ as follows:

1. Sample a value z from a distribution $g(z)$ from which sampling is easy, and for which

$$mg(z) > f(z) \quad m \text{ a constant} \tag{3}$$

   $mg(z)$ is called an envelope function because it envelops $f(z)$.

2. Compute the ratio

$$R = \frac{f(z)}{mg(z)} \tag{4}$$

3. Sample $u \sim Unif(0,1)$.

4. If $R > u$, then z is treated as a draw from $f(x)$. Otherwise return to step 1.

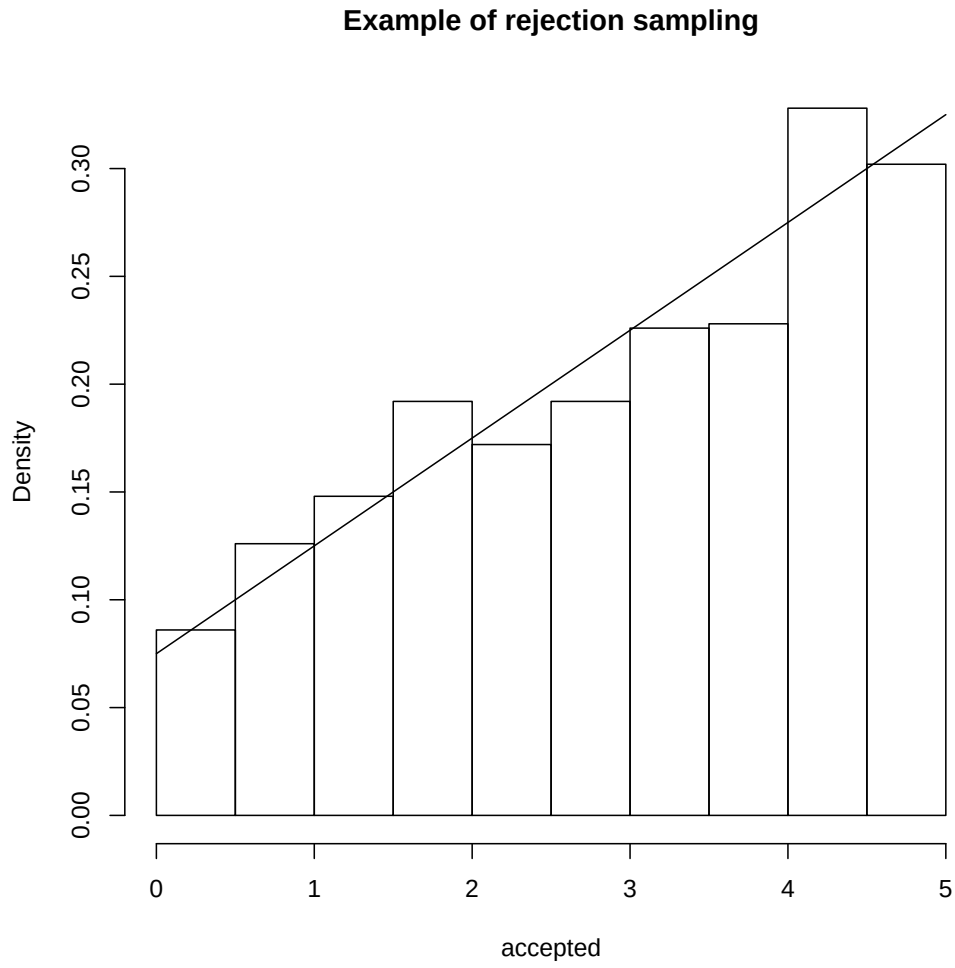
For example, consider f(x) to be: $f(x) = \frac{1}{40}(2x+3)$, with $0 < x < 5$. The maximum height of $f(x)$ is 0.325 (why? Try evaluating the function at x=5). So we need an envelope function that exceeds 0.325. The uniform density $Unif(0,5)$ has maximum height 0.2 (why is that?), so if we multiply it by 2 we have maximum height 0.4, which is greater than 0.325.

In the first step, we sample a number x from a uniform distribution Unif(0,5). This serves to locate a point on the x-axis between 0 and 5 (the domain of x). The next step involves locating a point in the y direction once the x coordinate is fixed. If we draw a number u from Unif(0,1), then $mg(x)u = 2 * 0.2u$ is a number between 0 and $2 * 0.2$. If this number is less than f(x), that means that the y value falls within f(x), so we accept it, else reject. Checking whether $mg(x)u$ is less than $f(x)$ is the same as checking whether

$$R = f(x)/mg(z) > u \tag{5}$$

```
#R program for rejection method of sampling
## From Lynch book, adapted by SV.
count<-0
k<-1
accepted<-rep(NA,1000)
rejected<-rep(NA,1000)
while(k<1001)
{
z<-runif(1,min=0,max=5)
r<-((1/40)*(2*z+3))/(2*.2)
if(r>runif(1,min=0,max=1)) {
  accepted[k]<-z
  k<-k+1} else {
    rejected[k]<-z
  }
count<-count+1
}
```

```r
hist(accepted,freq=F,
     main="Example of rejection sampling")

fn<-function(x){
  (1/40)*(2*x+3)
}

x<-seq(0,5,by=0.01)

lines(x,fn(x))
```



Here is the acceptance rate:

```
## acceptance rate:
table(is.na(rejected))[2]/
  sum(table(is.na(rejected)))

##  TRUE
## 0.538
```

Question: If you increase m, will acceptance rate increase or decrease? Stop here and come up with an answer before you read further.

Rejection sampling can be used with multivariate distributions.

Some limitations of rejection sampling: finding an envelope function may be difficult; the acceptance rate would be low if the constant m is set too high and/or if the envelope function is too high relative to f(x), making the algorithm inefficient.

5 The Gibbs sampling algorithm

Let Θ be a vector of parameter values, let length of Θ be k. Let j index the j-th iteration.

Algorithm:

1. Assign starting values to Θ:
   $\Theta^{j=0} \leftarrow S$
2. Set $j \leftarrow j + 1$
3. 1. Sample $\theta_1^j \mid \theta_2^{j-1} \dots \theta_k^{j-1}$.
   2. Sample $\theta_2^j \mid \theta_1^j \theta_3^{j-1} \dots \theta_k^{j-1}$.

⋮

   k. Sample $\theta_k^j \mid \theta_1^j \dots \theta_{k-1}^j$.
4. Return to step 1.

Example: Consider the bivariate distribution:

$$f(x, y) = \frac{1}{28}(2x + 3y + 2) \tag{6}$$

We can analytically work out the conditional distributions (this needs some basic calculus, so you should just take the results below on trust; if you know calculus, you can derive the conditionals from the joint distribution):

$$f(x \mid y) = \frac{f(x,y)}{f(y)} = \frac{(2x+3y+2)}{6y+8} \tag{7}$$

$$f(y \mid x) = \frac{f(x,y)}{f(x)} = \frac{(2x+3y+2)}{4y+10} \tag{8}$$

The Gibbs sampler algorithm is:

1. Set starting values for the two parameters $x = -5, y = -5$. Set j=0.
2. Sample x^{j+1} from $f(x \mid y)$ using inversion sampling. You need to work out the inverse of $f(x \mid y)$ and $f(y \mid x)$ first. To do this, for $f(x \mid u)$, we have find z_1:

$$u = \int_0^{z_1} \frac{(2x+3y+2)}{6y+8}\, dx \tag{9}$$

And for $f(y \mid x)$, we have to find z_2:

$$u = \int_0^{z_2} \frac{(2x+3y+2)}{4y+10}\, dy \tag{10}$$

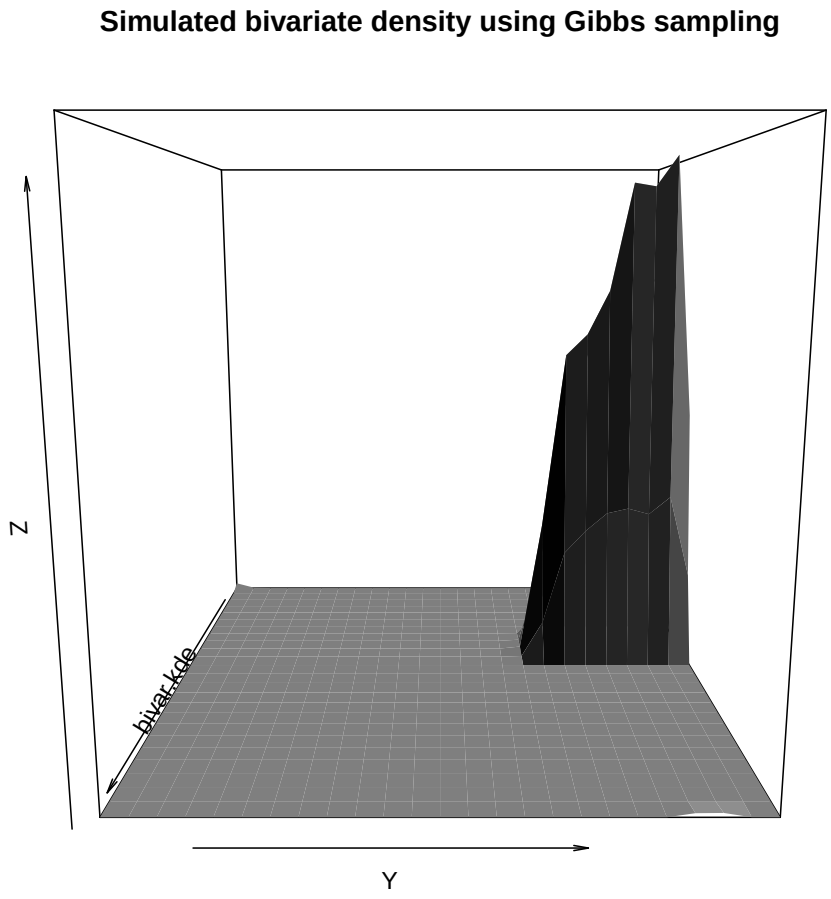
I leave that as an exercise; the solution is given in the code below.

```
#R program for Gibbs sampling using inversion method
## program by Scott Lynch, modified by SV:
x<-rep(NA,2000)
y<-rep(NA,2000)
x[1]<- -5
y[1]<- -5

for(i in 2:2000)
{ #sample from x | y
  u<-runif(1,min=0, max=1)
  x[i]<-sqrt(u*(6*y[i-1]+8)+(1.5*y[i-1]+1)*(1.5*y[i-1]+1))-
```

```
    (1.5*y[i-1]+1)
  #sample from y | x
u<-runif(1,min=0,max=1)
y[i]<-sqrt((2*u*(4*x[i]+10))/3 +((2*x[i]+2)/3)*((2*x[i]+2)/3))-
    ((2*x[i]+2)/3)
}
```

You can run this code to visualize the simulated posterior distribution:

```
library(MASS)
bivar.kde<-kde2d(x,y)
persp(bivar.kde,phi=10,theta=90,shade=0,border=NA,
      main="Simulated bivariate density using Gibbs sampling")
```



A central insight here is that knowledge of the conditional distributions is enough to figure out (simulate from) the joint distribution, provided such a joint distribution exists.

6 Gibbs sampling using rejection sampling

Suppose the conditional distribution is not univariate–we might have conditional multivariate densities. Now we can't use inversion sampling. Another situation where we can't use inversion sampling is when F^{-1} can't be calculated even in one dimension (An example where the inversion sampling

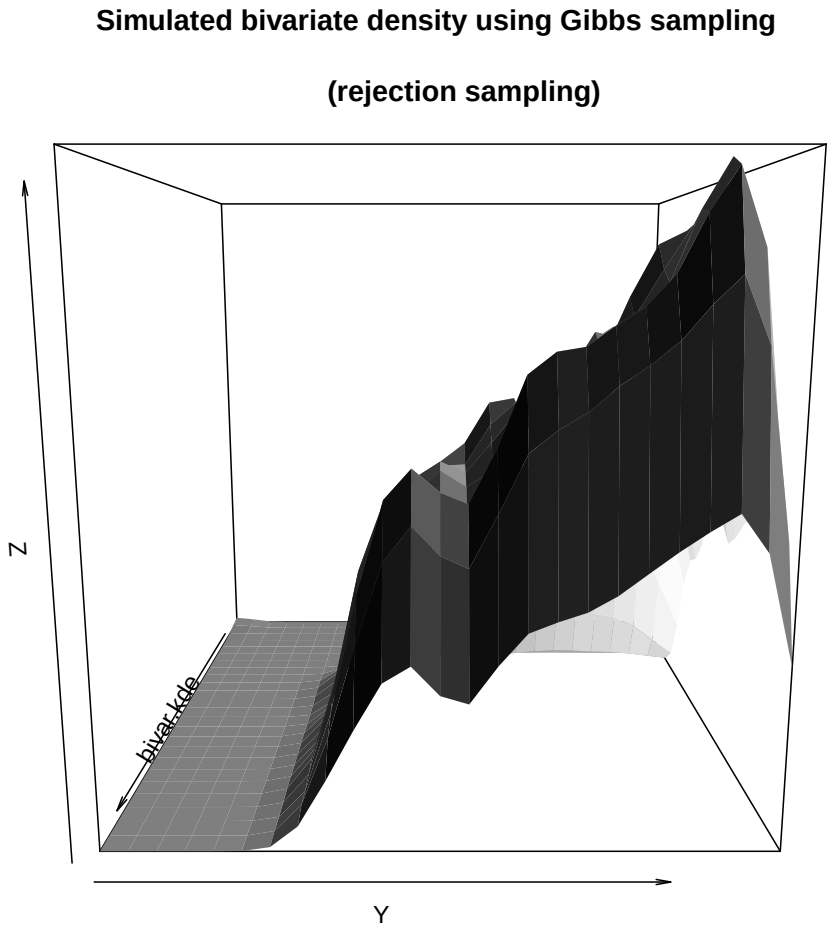
doesn't work is the bivariate normal; there is no way to compute the conditional CDF analytically).

```
#R program for Gibbs sampling using rejection sampling
## Program by Scott Lynch, modified by SV:
x<-rep(NA,2000)
y<-rep(NA,2000)
x[1]<- -1
y[1]<- -1


m<-25

for(i in 2:2000){
#sample from x | y using rejection sampling
  z<-0
  while(z==0){
  u<-runif(1,min=0, max=2)
  if( ((2*u)+(3*y[i-1])+2) > (m*runif(1,min=0,max=1)*.5))
    {
    x[i]<-u
    z<-1
  }
}
  #sample from y | x using z=0 while(z==0)
 z<-0
  while(z==0){
    u<-runif(1,min=0,max=2)
  if( ((2*x[i])+(3*u)+2)> (m*runif(1,min=0,max=1)*.5)){
{y[i]<-u; z<-1}
}
  }
}
```

Note that we need to know the conditional densities only up to proportionality, we do not need to know the normalizing constant (see discussion in Lynch). Also, note that we need sensible starting values, otherwise the algorithm will never take off.

Visualization:

```
bivar.kde<-kde2d(x,y)
persp(bivar.kde,phi=10,theta=90,shade=0,border=NA,
      main="Simulated bivariate density using Gibbs sampling \n
      (rejection sampling)")
```



7 More realistic example: Sampling from a bivariate density

Let's try to sample from a bivariate normal distribution using Gibbs sampling. We could have just done it with a built-in function from the `MASS` package for this (see earlier material in the lecture notes on bivariate distribution sampling using `mvrnorm`). But it's instructive to do it "by hand".

Here, we can compute the conditional distributions but we can't compute $F^{-1}()$ analytically.

We can look up in a textbook (or derive this analytically) that the conditional distribution $f(X \mid Y)$ in this case is:

$$f(X \mid Y) = N(\mu_x + \rho\sigma_x \frac{y - \mu_y}{\sigma_y}, \sigma_x\sqrt{(1 - \rho^2)}) \tag{11}$$

and similarly (mutatis mutandis) for $f(Y \mid X)$. Note that Lynch provides a derivation for conditional densities up to proportionality.

For simplicity we assume we have a bivariate normal, uncorrelated random variables X and Y each with mean 0 and sd 1. But you can play with all of the parameters below and see how things change.

Here is my code:

```
## parameters:
mu.x<-0
mu.y<-0
sd.x<-1
sd.y<-1
rho<-0

## Gibbs sampler:
x<-rep(NA,2000)
y<-rep(NA,2000)
x[1]<- -5
y[1]<- -5

for(i in 2:2000)
{ #sample from x | y, using (i-1)th value of y:
  u<-runif(1,min=0, max=1)
```

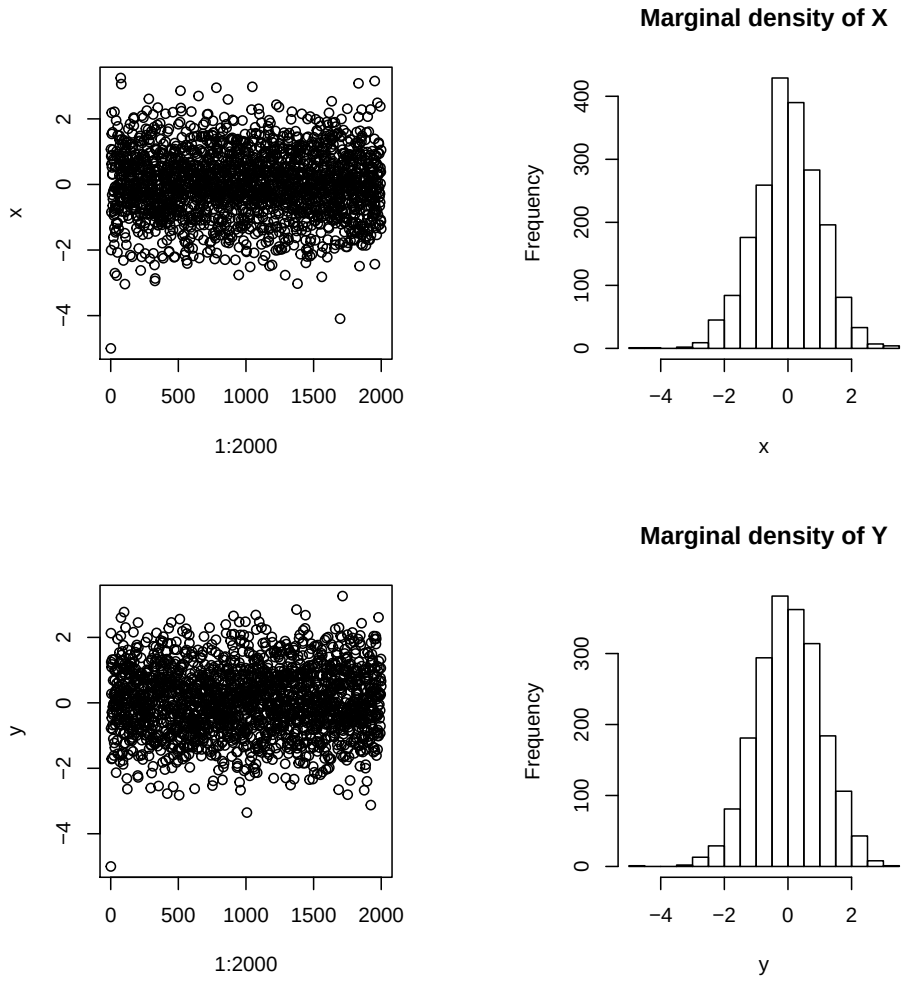
```
  x[i]<- rnorm(1,mu.x+rho*sd.x*((y[i-1] - mu.y)/sd.y),
               sd.x*sqrt(1-rho^2))

  #sample from y | x, using i-th value of x
u<-runif(1,min=0,max=1)
y[i]<-rnorm(1,mu.y+rho*sd.y*((x[i] - mu.x)/sd.x),
            sd.y*sqrt(1-rho^2))
}
```

We can visualize this as well. The code below rotates the angle of viewing (theta) through a for-loop, so you can view the posterior from different directions.

```
bivar.kde<-kde2d(x,y)
for(i in 1:100){
persp(bivar.kde,phi=10,theta=i,shade=0,border=NA,
      main="Simulated bivariate normal density
      using Gibbs sampling")
Sys.sleep(0.1)
}
```

We can plot the "trace plot" of each density, as well as the marginal density:

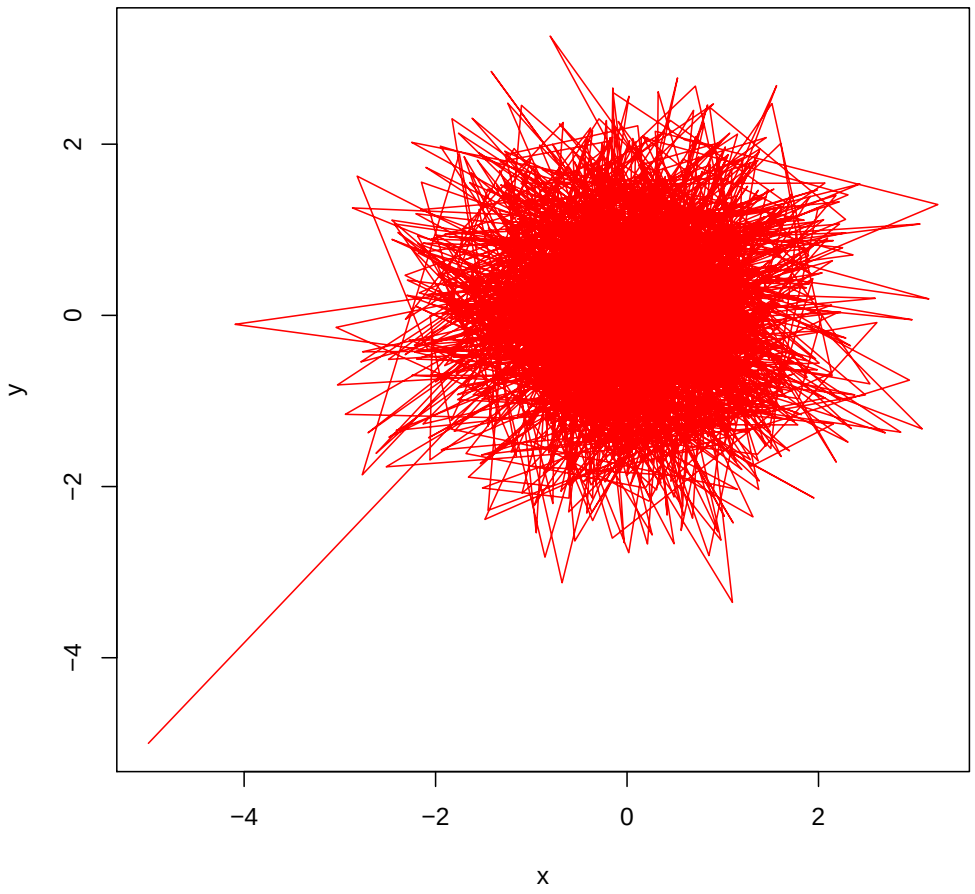
```
op<-par(mfrow=c(2,2),pty="s")
plot(1:2000,x)
hist(x,main="Marginal density of X")
plot(1:2000,y)
hist(y,main="Marginal density of Y")
```



The trace plots show the points that were sampled. The initial values (-5) were not realistic, and it could be that the first few iterations do not really yield representative samples. We discard these, and the initial period is called "burn-in" or (in Stan) "warm up".

The two dimensional trace plot traces the Markov chain walk:

```
plot(x,y,type="l",col="red")
```



We can also summarize the marginal distributions. One way is to compute the 95% credible interval, and the mean or median.

```
quantile(x,0.025)
```

```
##     2.5%
## -2.0589
```

```
quantile(x,0.975)
```

```
##   97.5%
## 1.9525
```

```
mean(x)

## [1] -0.0043403

quantile(y,0.025)

##    2.5%
## -1.9675

quantile(y,0.975)

## 97.5%
## 2.016

mean(y)

## [1] 0.033606
```

These numbers match up pretty well with the theoretical values (which we know since we sampled from a bivariate with known means and sds; see above).

If we discard the first 500 runs as burn-in, we get:

```
quantile(x[501:2000],0.025)

##    2.5%
## -1.9833

quantile(x[501:2000],0.975)

##  97.5%
## 1.9414

mean(x[501:2000])

## [1] -0.00104

quantile(y[501:2000],0.025)
```

```
##    2.5%
## -1.9643

quantile(y[501:2000],0.975)

##  97.5%
## 2.0253

mean(y[501:2000])

## [1] 0.030101
```

8 Sampling the parameters given the data

The Bayesian approach is that, conditional on having data and priors on parameters, we want to sample the posterior distributions of the parameters. For this, we need to figure out the conditional density of the parameters given the data.

The marginal for σ^2 happens to be:

$$p(\sigma^2 \mid X) \propto InverseGamma((n-1)/2, (n-1)var(x)/2) \tag{12}$$

The conditional distribution:

$$p(\sigma^2 \mid \mu, X) \propto InverseGamma(n/2, \sum (x_i - \mu)^2/2) \tag{13}$$

$$p(\mu \mid \sigma^2, \mid X) \propto N(\bar{x}, \sigma^2/n) \tag{14}$$

Now we can do Gibbs sampling on parameters given data. There are two ways, given the above equations:

1. Given the marginal distribution of σ^2, sample a vector of values for σ^2 and then sample μ conditional on each value of σ^2 from μ's conditional distribution. **This approach is more efficient**.

2. First sample a value for σ^2 conditional on μ, then sample a value of μ conditional on the new value for σ^2, etc.

To illustrate, suppose we have a vector of scores, x.

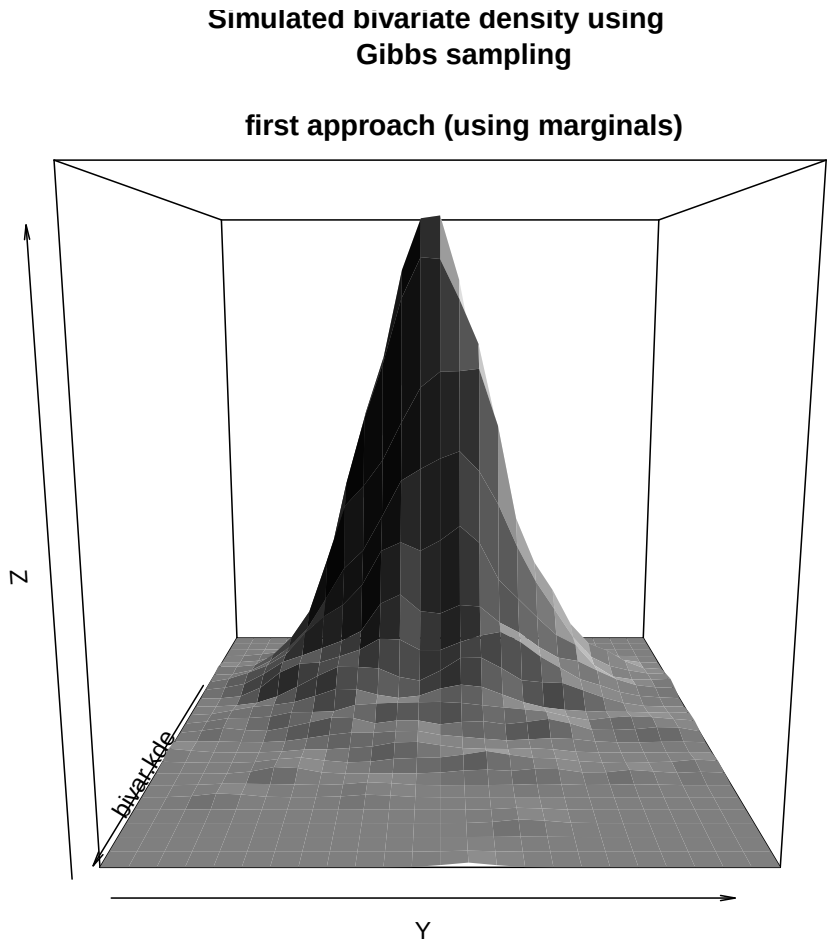
8.1 Example of first approach

```
#R: sampling from marginal for variance and conditional for mean
## code by Scott Lynch, slightly modified by SV:
x<-rnorm(20,mean=12,sd=2)
sig2<-rgamma(2000,(length(x)-1)/2 ,
             rate=((length(x)-1)*var(x)/2))
sig2<-1/sig2
mu<-rnorm(2000,mean=mean(x),
          sd=(sqrt(sig2/length(x))))
```

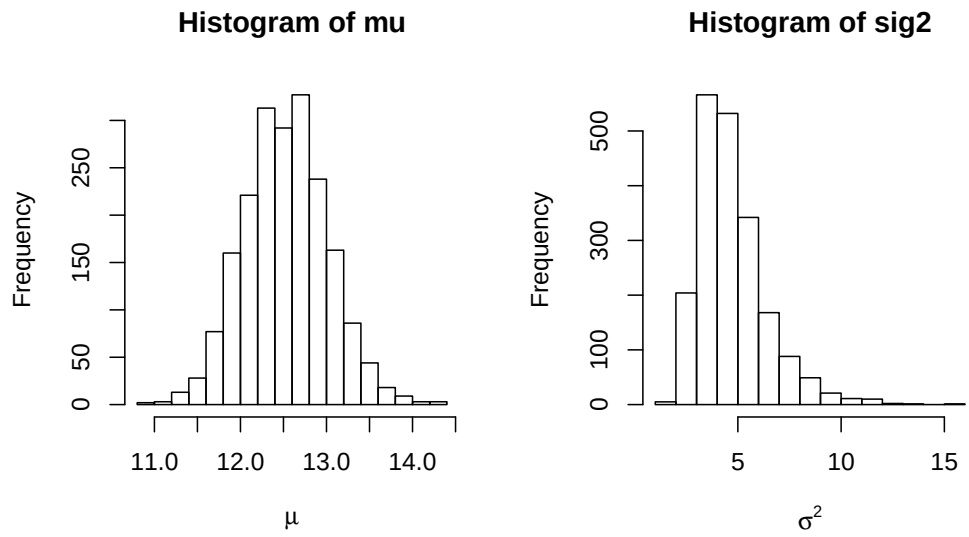
Note that we draw from a Gamma, and then invert to get an inverse Gamma.

Visualization:

```
bivar.kde<-kde2d(sig2,mu)
persp(bivar.kde,phi=10,theta=90,shade=0,
      border=NA,
      main="Simulated bivariate density using
      Gibbs sampling\n
      first approach (using marginals)")
```




```
op<-par(mfrow=c(1,2),pty="s")
hist(mu,xlab=expression(mu))
hist(sig2,xlab=expression(sigma^2))
```



Example of second approach Here, we sample μ and σ^2 sequentially from their conditional distributions.

```
#R: sampling from conditionals for both variance and mean
mu<-rep(0,2000)
sig2<-rep(1,2000)
for(i in 2:2000){
  sig2[i]<-rgamma(1,(length(x)/2),
                  rate=sum((x-mu[i-1])^2)/2)
```

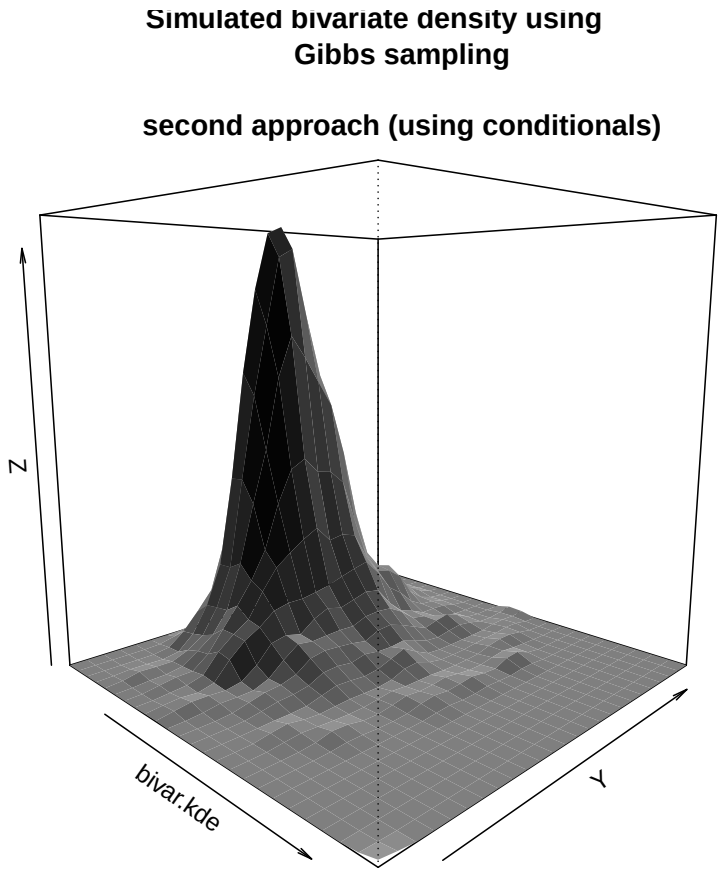
```
  sig2[i]<-1/sig2[i]
  mu[i]<-rnorm(1,mean=mean(x),
               sd=sqrt(sig2[i]/length(x)))
}
```

First, we drop the burn-in values (in the conditionals sampling approach, the initial values will need to be dropped).

```
mu<-mu[1001:2000]
sig2<-sig2[1001:2000]
```

Visualization:

```
bivar.kde<-kde2d(sig2,mu)
persp(bivar.kde,phi=10,theta=45,shade=0,border=NA,
      main="Simulated bivariate density using
      Gibbs sampling\n
      second approach (using conditionals)")
```



9 Summary of the story so far

This summary is taken, essentially verbatim but reworded a bit, from Lynch (pages 103-104).

1. Our main goal in the Bayesian approach is to summarize the posterior density.

2. If the posterior density can be analytically analyzed using integration, we are done.

3. If there are no closed-form solutions to the integrals, we resort to sampling from the posterior density. This is where Gibbs sampling comes in: the steps are:

   (a) Derive the relevant conditional densities: this involves (a) treating other variables as fixed (b) determining how to sample from the resulting conditional density. The conditional density either has a known form (e.g., normal) or it may take an unknown form. If it has an unknown form, we use inversion or rejection sampling in order to take samples from the conditional densities.

   (b) If inversion sampling from a conditional density is difficult or impossible, we use the **Metropolis-Hastings algorithm**, which we discuss next.

10 Metropolis-Hastings sampling

I got this neat informal presentation of the idea from
http://www.faculty.biol.ttu.edu/strauss/bayes/LectureNotes/09_MetropolisAlgorithm.pdf
Quoted almost verbatim (slightly edited):

1. Imagine a politician visiting six islands:

   (a) Wants to visit each island a number of times proportional to its population.

   (b) Doesn't know how many islands there are.

   (c) Each day: might move to a new neighboring island, or stay on current island.

2. Develops simple heuristic to decide which way to move:

   (a) Flips a fair coin to decide whether to move to east or west.

   (b) If the proposed island has a larger population than the current island, moves there.

   (c) If the proposed island has a *smaller* population than the current island, moves there probabilistically, based on uniform distribution.

Probability of moving depends on relative population sizes:

$$p_{move} = \frac{p_{proposed}}{p_{current}} \tag{15}$$

3. End result: probability that politician is on any one of the islands exactly matches the relative proportions of people on the island. We say that the probability distribution **converges** to a particular distribution.

More formal presentation of Metropolis-Hastings One key point here is that in MH, we need only specify the unnormalized joint density for the parameters, we don't need the conditional densities. One price to be paid is greater computational overhead.

As Lynch (page 108) puts it:

> "A key advantage to the MH algorithm over other methods of sampling, like inversion and rejection sampling, is that it will work with multivariate distributions (unlike inversion sampling), and we do not need an enveloping function (as in rejection sampling)."

The algorithm (taken almost verbatim from Lynch):

1. Establish starting values S for the parameter: $\theta^{j=0} = S$, using MLE or some other method (apparently the starting values don't matter—the distribution to which the Markov chain converges will be the posterior distribution of interest—but poor starting values will affect speed of convergence). Set $j = 1$.

2. Draw a "candidate" parameter, θ^c from a "proposal density," $\alpha(\cdot)$.

3. Compute the ratio:

$$R = \frac{f(\theta^c)}{f(\theta^{j-1})} \frac{\alpha(\theta^{j-1} \mid \theta^c)}{\alpha(\theta^c \mid \theta^{j-1})} \tag{16}$$

   The first part of the ratio ($\frac{f(\theta^c)}{f(\theta^{j-1})}$) is called the **importance ratio**: the ratio of the unnormalized posterior density evaluated at the candidate parameter value (θ^c) to the posterior density evaluated at the previous parameter value (θ^{j-1}).

The second part of the ratio is the ratio of the **proposal** densities evaluated at the candidate and previous points. This ratio adjusts for the fact that some candidate values may be selected more often than others.

4. Compare R with a $Unif(0,1)$ random draw u. If $R > u$, then set $\theta^j = \theta^c$. Otherwise, set $\theta^j = \theta^{j-1}$.

   An equivalent way of saying this: set an **acceptance probability** *aprob* as follows:

$$aprob = min(1, R) \tag{17}$$

   Then sample a value u from a uniform $u \sim Unif(0,1)$.

   If $u < aprob$, then accept candidate, else stay with current value (x_{i-1}). What this means is that if aprob is larger than 1, then we accept the candidate, and if it is less than 1, then we accept the candidate with probability *aprob*.

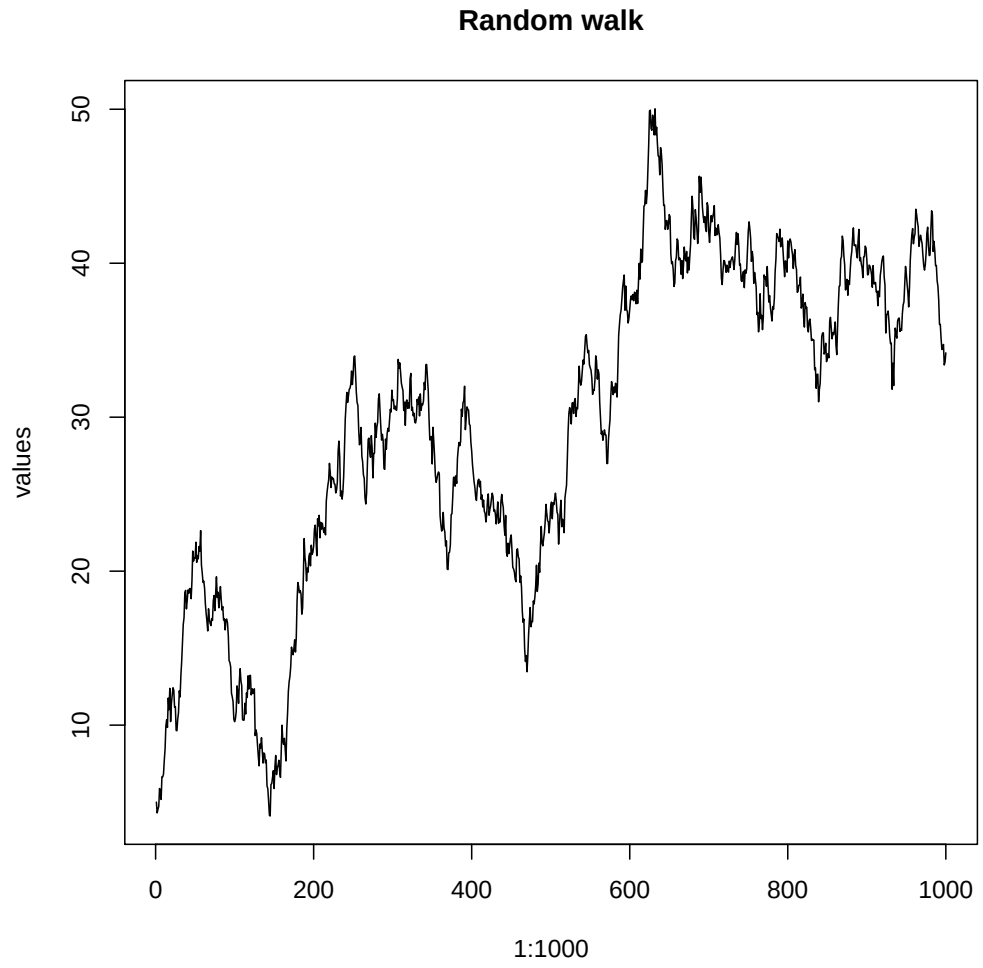
5. Set $j = j + 1$ and return to step 2 until enough draws are obtained.

Step 2 is like drawing from an envelope function in rejection sampling, except that the proposal density (which is any density that it's easy to sample from) does not have to envelope the density of interest. As in rejection sampling, we have to check whether the draw of a parameter value from this proposal density can be considered to be from the density of interest. For example, a candidate value could be drawn from a normal distribution $\theta^c = \theta^{j-1} + N(0, c)$, where c is a constant. This gives us a "random walk Metropolis algorithm".

We can quickly implement such a random walk to get a feel for it. This is one of the things the MH algorithm will be doing.

```
## implementation of random walk Metropolis:
theta.init<-5
values<-rep(NA,1000)
values[1]<-theta.init
for(i in 2:1000){
  values[i]<-values[i-1]+rnorm(1,mean=0,sd=1)
}
```

```
plot(1:1000,values,type="l",main="Random walk")
```



And here is an example of a draw from a proposal density: Suppose that at the $(j-1)$-th step, we have the value θ^{j-1} (in the very first step, when j=2, we will be using the starting value of θ). Suppose that our proposal density is a normal. Our candidate draw will then be made using θ^{j-1}: $\theta^c = \theta^{j-1} + N(0, c)$, c some constant. Suppose $\theta^{j-1} = 5$, and $c = 1$. We can draw a candidate value θ^c as follows:

```
(theta.c<-5 + rnorm(1,mean=0,sd=1))

## [1] 6.1362
```

This is now our θ^c.

For Step 3 in the algorithm, we can now calculate:

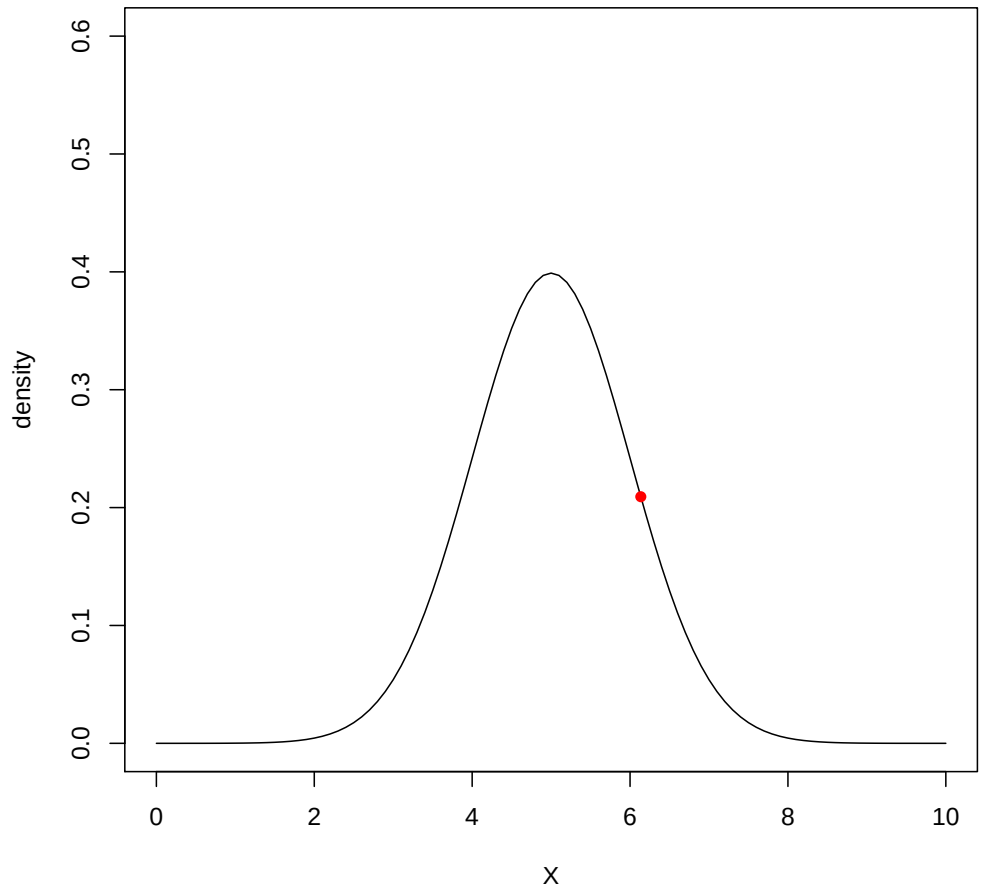
$$\alpha(\theta^c \mid \theta^{j-1}) \tag{18}$$

as follows:

```
(p1<-dnorm(theta.c,mean=5,sd=1))

## [1] 0.20921
```

Here is a visualization of what the above call does: given the candidate θ^c, we are computing the probability of transitioning to it (the probability of it being the next position moved to on the x-axis) given a normal distribution with mean 5, which is our starting value for x (sd of the normal distribution is 1, for simplicity).

```
plot(function(x) dnorm(x,mean=5), 0, 10,
    ylim=c(0,0.6),
    ylab="density",xlab="X")
points(theta.c,p1,pch=16,col="red")
```



We can also calculate

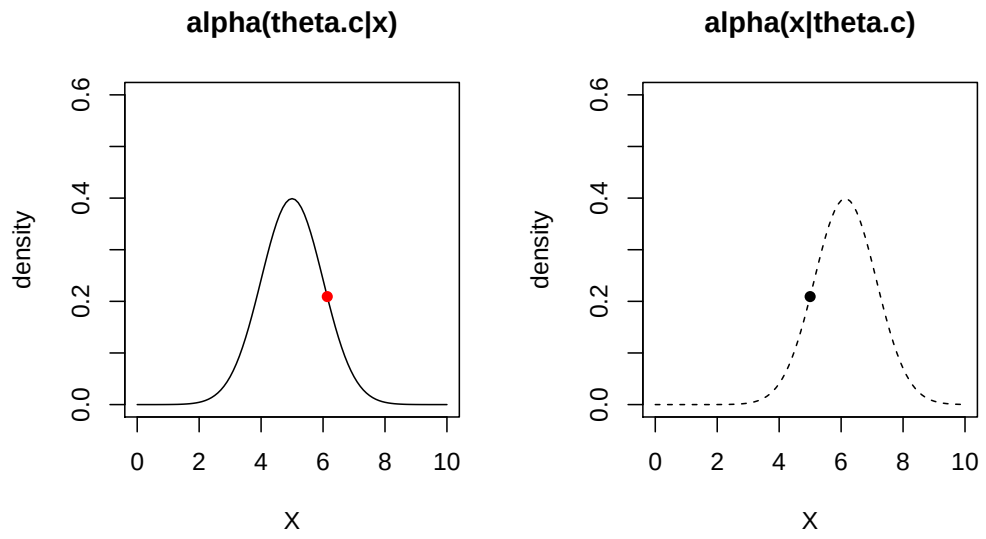
$$\alpha(\theta^{j-1} \mid \theta^c) \tag{19}$$

as follows:

```
(p2<-dnorm(5,mean=theta.c,sd=1))

## [1] 0.20921
```

Let's visualize what is being computed here. First I re-plot the above visualization for comparison.

```
op<-par(mfrow=c(1,2),pty="s")
plot(function(x) dnorm(x,mean=5), 0, 10,
    ylim=c(0,0.6),
    ylab="density",xlab="X",main="alpha(theta.c|x)")
points(theta.c,p1,pch=16,col="red")

plot(function(x) dnorm(x,mean=theta.c), 0, 10,
    ylim=c(0,0.6),
    ylab="density",xlab="X",new=F,lty=2,,main="alpha(x|theta.c)")
points(5,p2,pch=16,col="black")
```



These two calculations allows us to compute the ratio $\frac{\alpha(\theta^{j-1}|\theta^c)}{\alpha(\theta^c|\theta^{j-1})}$ (which will turn up in the next step):

```
p2/p1
```

```
## [1] 1
```

Note that the ratio is 1. If you look at the figure above, you will see that for symmetric distributions (here, the normal distribution) this ratio will **always** be 1. In other words, for symmetric proposal distributions, we can ignore the second term in the calculation of R and focus only on the

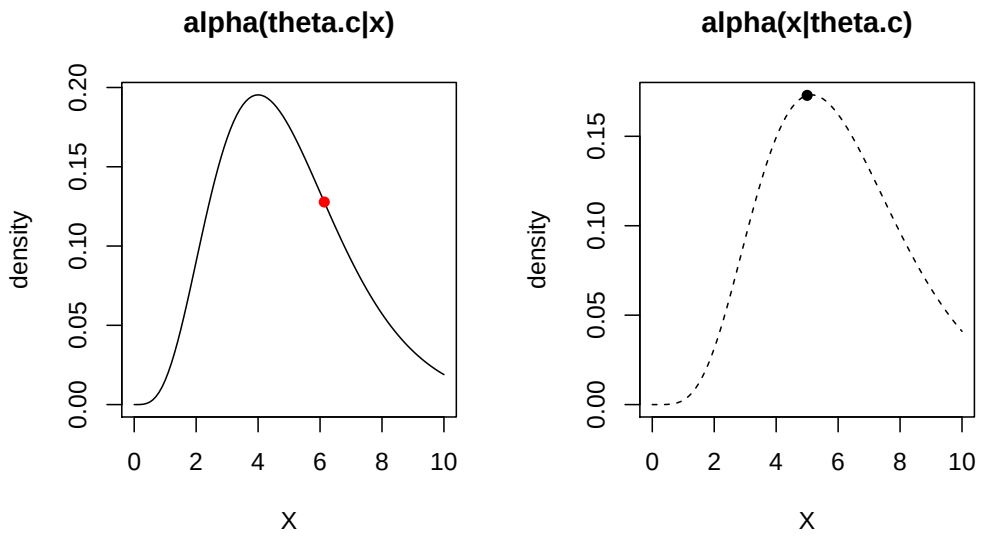
importance ratio (see above).

What does it mean for the p2/p1 ratio to be 1? It means is that the probability of transitioning to θ^{j-1} when sampling from $N(\theta^c, 1)$ is the same as the probability of transitioning to θ^c when sampling from $N(\theta^{j-1}, 1)$. (Variance is set at 1 just for illustration; one could choose a different value.)

The ratio p2/p1 becomes relevant for asymmetric proposal densities. Lynch writes: "This ratio adjusts for the fact that, with asymmetric proposals, some candidate values may be selected more often than others..." In such situations p1 would be larger than p2, and so p2/p1 will be smaller than 1. Adding this term to the calculation of R down-adjusts the value of R.

Let's try to visualize this. What we see below is that the probability of transitioning to the candidate is higher given x (LHS plot) than the probability of transitioning to x given the candidate (RHS).

```
op<-par(mfrow=c(1,2),pty="s")
plot(function(x) dgamma(x,shape=5), 0, 10,
    #ylim=c(0,0.6),
    ylab="density",xlab="X",main="alpha(theta.c|x)")
p1<-dgamma(theta.c,shape=5)
points(theta.c,p1,pch=16,col="red")

plot(function(x) dgamma(x,shape=theta.c), 0, 10,
    #ylim=c(0,0.6),
    ylab="density",xlab="X",
     lty=2,,main="alpha(x|theta.c)")
p2<-dgamma(5,shape=theta.c)
points(5,p2,pch=16,col="black")
```



This asymmetry is what the second term in the ratio R corrects for.

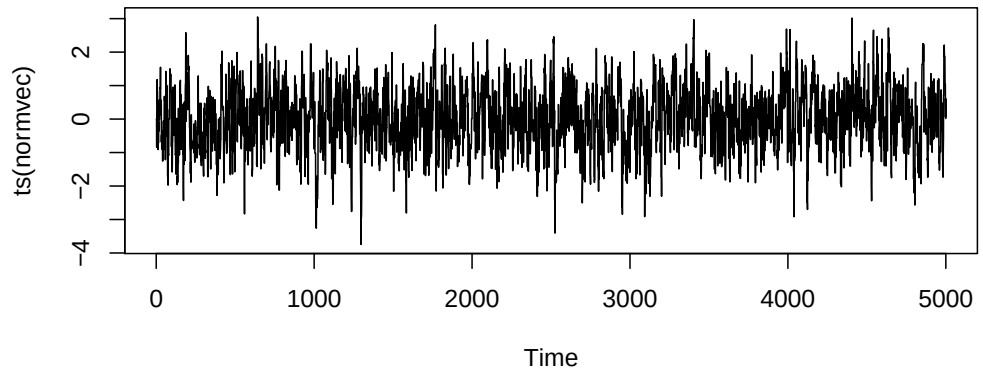
When the second term is needed (when it's 1 because we have a symmetric proposal), we have the Metropolis algorithm. When the second term is needed, we have the Metropolis-Hastings algorithm.

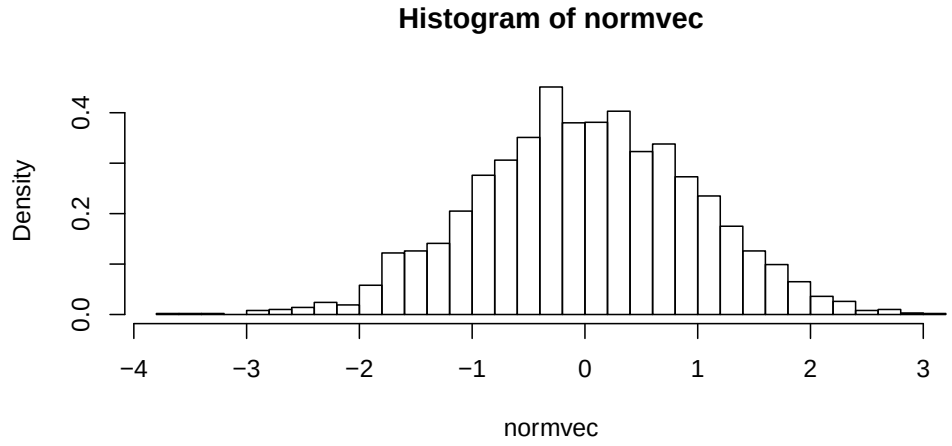
11 Implementing and visualizing the MH algorithm

Here is a simple implementation of a Metropolis algorithm, which involves such a symmetric situation:

```
## source:
##http://www.mas.ncl.ac.uk/~ndjw1/teaching/sim/metrop/metrop.html
## slightly adapted by SV:
#n is num. of simulations
norm<-function (n)
{
        vec <- vector("numeric", n)
        x <- 0
        vec[1] <- x
        for (i in 2:n) {
                can <- x + rnorm(1,mean=0,sd=1)
                ## to-do need to anticipate this in notes:
                aprob <- min(1, dnorm(can)/dnorm(x))
                u <- runif(1)
                if (u < aprob){
                        x <- can
                }
                vec[i] <- x
        }
        vec
}
```

```
normvec<-norm(5000)
op<-par(mfrow=c(2,1))
plot(ts(normvec))
hist(normvec,30,freq=F)
```






```
## we get the correct posterior distribution:
mean(normvec)

## [1] 0.0081252

sd(normvec)

## [1] 0.96726
```

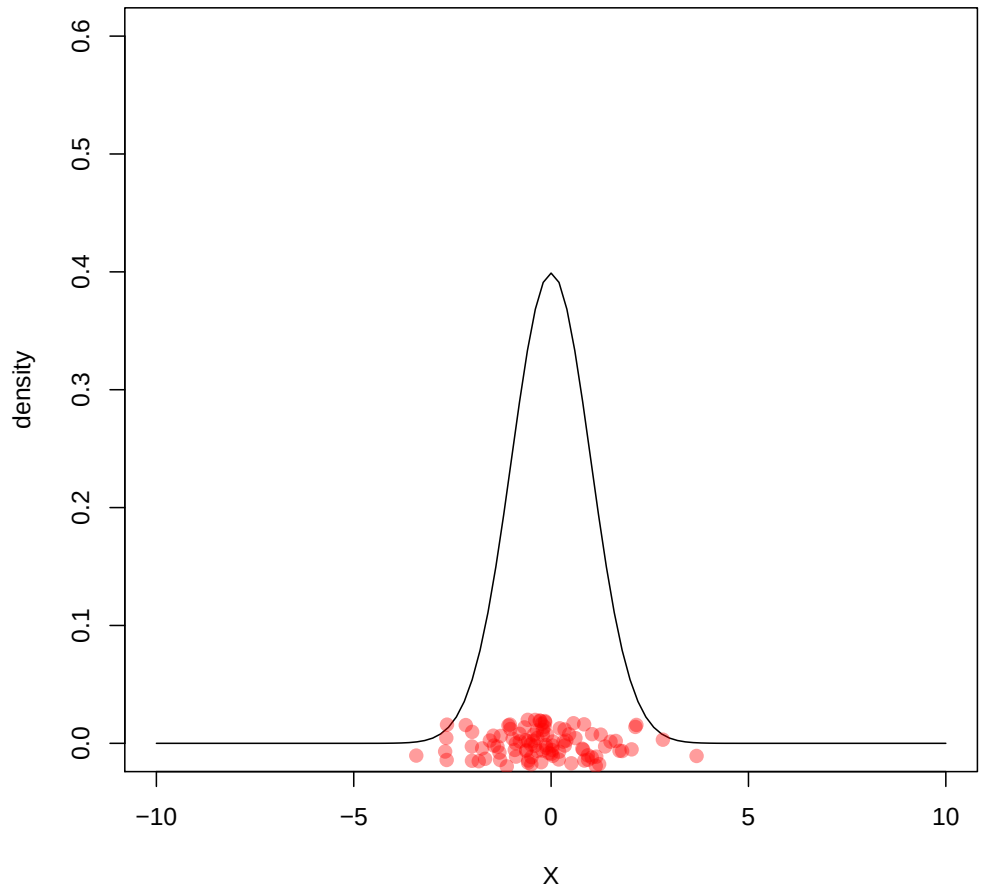
Visualization of the unfolding move-no move decisions I do this in two ways. One shows how the choice between the candidate and the current x is resolved, and the other shows the end result of the algorithm's run.

I first set up a function to create transparent colors (there is probably a better way than this, need to look into that):

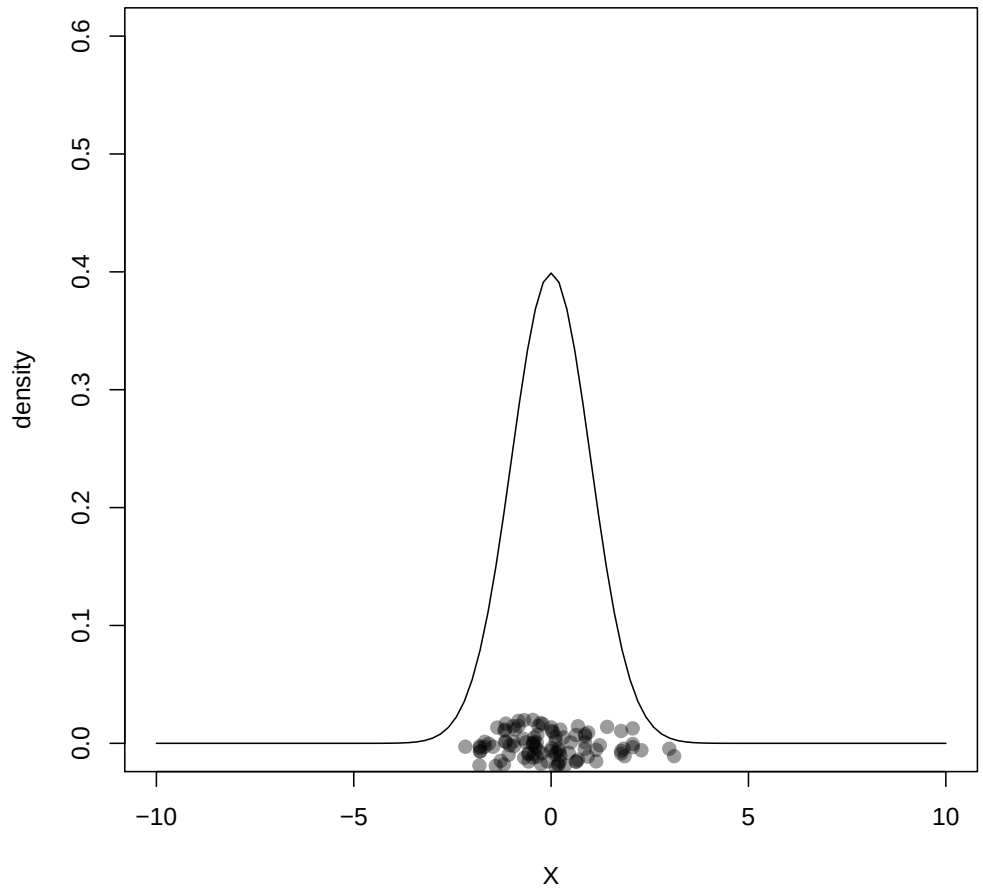
```
## I got this from the internet somewhere:
addTrans <- function(color,trans)
{
  # This function adds transparency to a color.
  # Define transparency with an integer between 0 and 255
  # 0 being fully transperant and 255 being fully visible
  # Works with either color and trans a vector of equal length,
  # or one of the two of length 1.

  if (length(color)!=length(trans)&!any(c(length(color),
                                          length(trans))==1))
    stop("Vector lengths not correct")
  if (length(color)==1 & length(trans)>1) color <-
    rep(color,length(trans))
  if (length(trans)==1 & length(color)>1) trans <-
    rep(trans,length(color))

  num2hex <- function(x)
  {
    hex <- unlist(strsplit("0123456789ABCDEF",split=""))
    return(paste(hex[(x-x%%16)/16+1],hex[x%%16+1],sep=""))
  }
  rgb <- rbind(col2rgb(color),trans)
  res <- paste("#",apply(apply(rgb,2,num2hex),2,
                         paste,collapse=""),sep="")
  return(res)
}
```

First, the incremental presentation, with 100 steps:

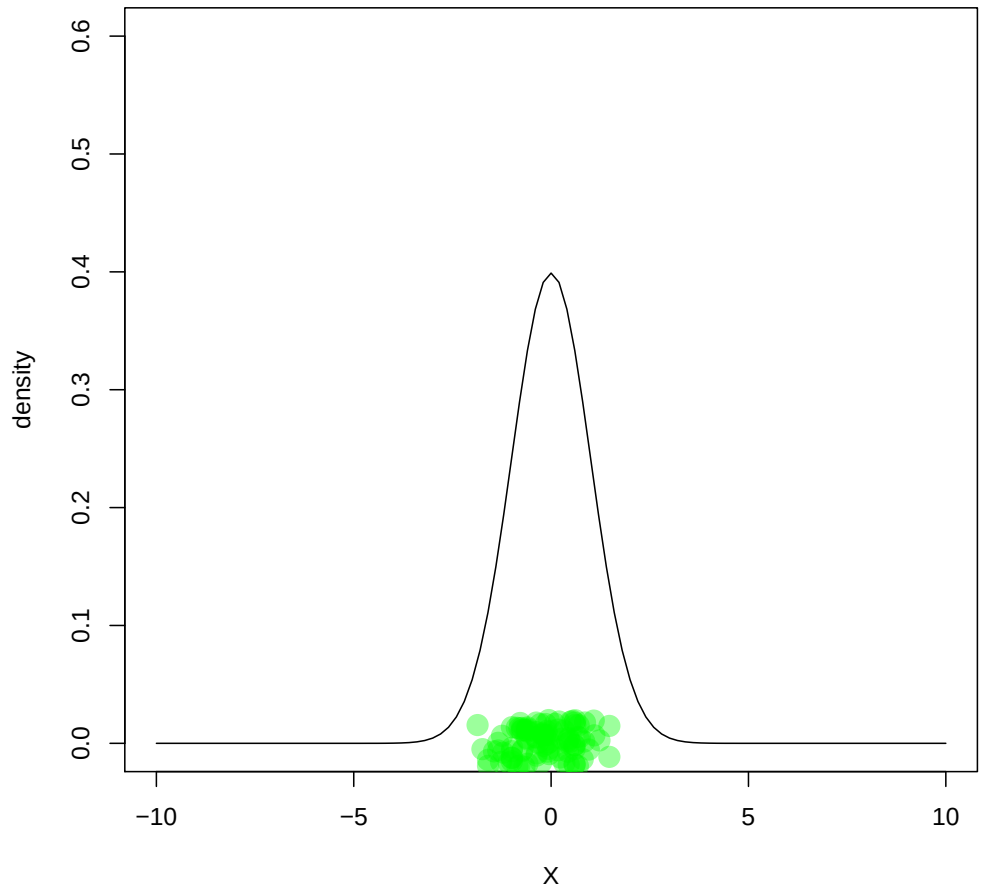
```
n<-100
norm<-function (n,display="candidate")
{
        vec <- vector("numeric", n)
        x <- 0
        vec[1] <- x
        plot(function(x) dnorm(x), -10, 10,
        ylim=c(0,0.6),
        ylab="density",xlab="X")
        for (i in 2:n) {
               if(display=="x"){
                points(x,jitter(0),pch=16,
                       col=addTrans("black",100),cex=1.3)}
                can <- x + rnorm(1,mean=0,sd=1)
                aprob <- min(1, dnorm(can)/dnorm(x))
                u <- runif(1)
                if (u < aprob){
                        x <- can
                }
                vec[i] <- x
               if(display=="candidate"){
                points(can,jitter(0),pch=16,
                       col=addTrans("red",100),cex=1.3)}
#                 Sys.sleep(1)
               if(display=="choice"){
                points(vec[i],jitter(0),pch=16,
                       col=addTrans("green",100),cex=2)}
                Sys.sleep(0.5)
        }
        vec
}

#op<-par(mfrow=c(1,3),pty="s")
normvec<-norm(n=100,display="candidate")
```




```
normvec<-norm(n=100,display="x")
```




```
normvec<-norm(n=100,display="choice")
```



The red dots are the candidates, the black the X's, and the green ones are ones are the choices adopted at each step.

The next visualization shows the end result of the algorithm run:

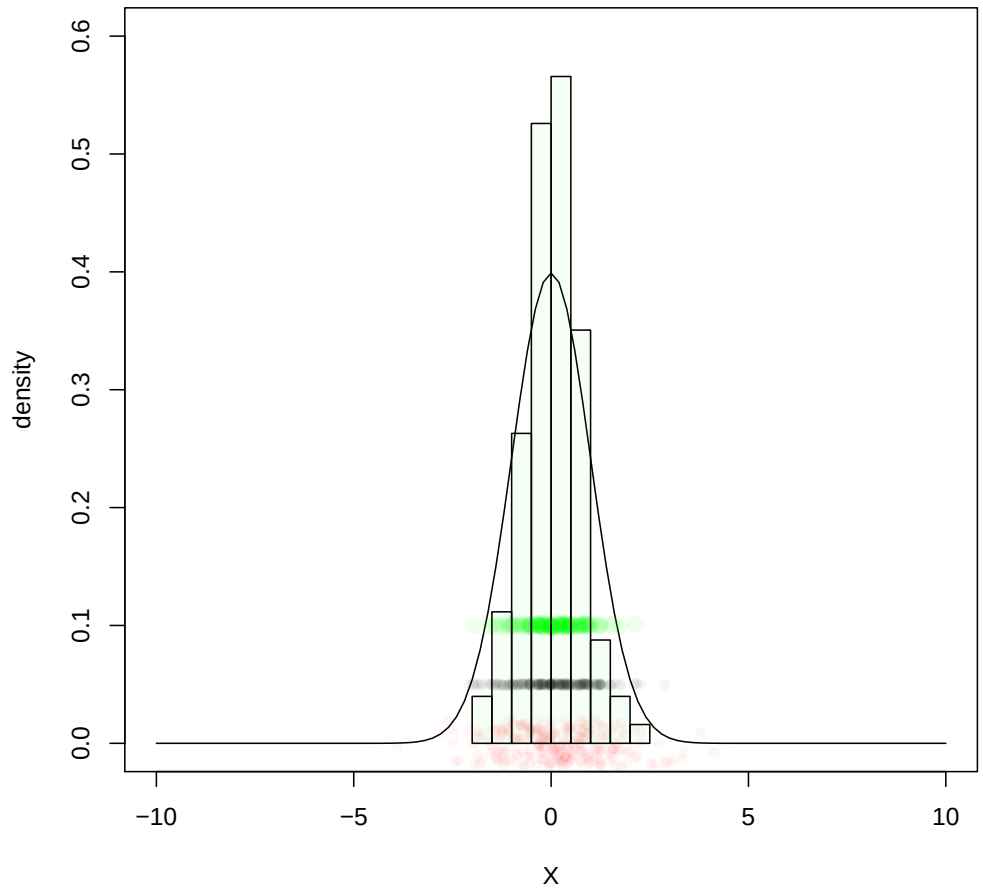
```
n<- 500
can.store<-rep(NA,n)
x.store<-rep(NA,n)
norm<-function (n)
{
        vec <- vector("numeric", n)
        x <- 0
```

```
        vec[1] <- x
        for (i in 2:n) {
                can <- x + rnorm(1,mean=0,sd=1)
                can.store[i]<-can
                x.store[i]<-x
                aprob <- min(1, dnorm(can)/dnorm(x))
                u <- runif(1)
                if (u < aprob){
                        x <- can
                }
                vec[i] <- x
        }
        list(vec,can.store,x.store)
}
```

```
normvec<-norm(n)

## recover history of candidates and x's:
## discard first half:
start<-n/2
can.store<-normvec[2][[1]][start:n]
x.store<-normvec[3][[1]][start:n]

plot(function(x) dnorm(x), -10, 10,
                  ylim=c(0,0.6),
                  ylab="density",xlab="X")
#ys<-seq(0,.4,by=0.4/(n/2))
points(can.store,jitter(rep(0,length(can.store))),cex=1,pch=16,
       col=addTrans(rep("red",length(can.store)),10))
points(x.store,jitter(rep(0.05,length(x.store))),cex=1,pch=16,
       col=addTrans(rep("black",length(can.store)),10))
#Sys.sleep(2)
#                 plot(function(x) dnorm(x,mean=can), -10, 10,
#                   ylim=c(0,1),
#                   ylab="density",xlab="X",add=T,col="red")
cand.prob<-dnorm(can.store)
```

```
x.prob<-dnorm(x.store)
## cases where the candidate is chosen:
move<-runif(1)<cand.prob/x.prob
## at which steps did we take the candidates or stayed at x:
indices<-c(which(move),which(!move))
## erase earlier candidates:
#points(can.store,ys,cex=2,pch=16,col="white")
#points(x.store,ys,cex=2,pch=16,col="white")

## redraw target dist:
#plot(function(x) dnorm(x), -10, 10,
#                   ylim=c(0,0.6),
#                   ylab="density",xlab="X")


move.history<-data.frame(indices=indices,
              points=c(can.store[which(move)],
                       x.store[which(!move)]))

## reordered to reflect history:
move.history<-move.history[order(move.history[1,])]

points(move.history$points,
       jitter(rep(0.1,length(move.history$points))),
       pch=16,
       col=addTrans(rep("green",
                        length(move.history$points)),10),
       cex=1.5)

#legend(5,.6,pch=16,col=c("red","black"),
#       legend=c(paste("candidate",round(cand.prob,digits=2)),
#                paste("previous",round(x.prob,digits=2))))

hist(move.history$points,freq=FALSE,add=T,
     col=addTrans(rep("green",length(move.history$points)),10))
```

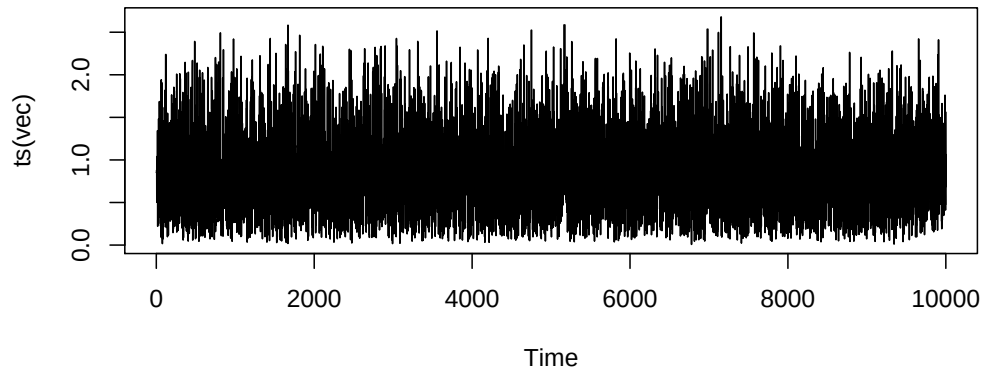
```
## Warning in if (!add) {: the condition has length > 1 and only
the first element will be used
```

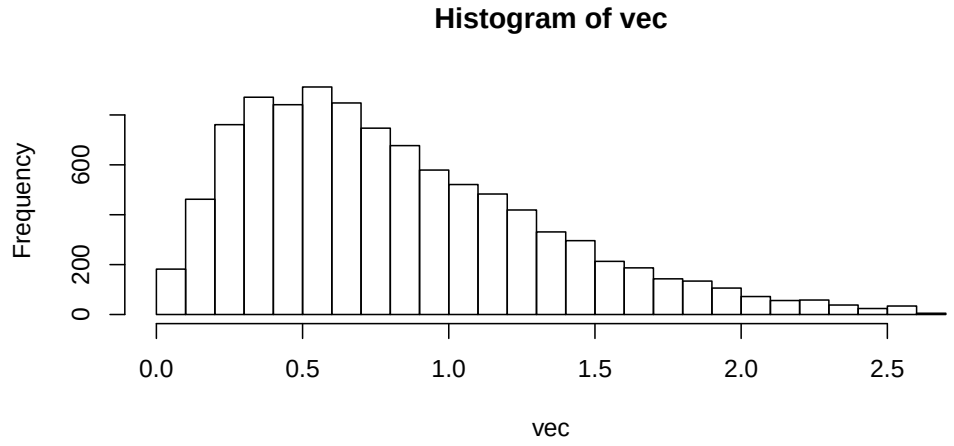


When we have an asymmetry, we use the Metropolis-Hastings algorithm:

```
## source: http://www.mas.ncl.ac.uk/~ndjw1/teaching/sim/metrop/indep.r
# metropolis-hastings independence sampler for a
# gamma based on normal candidates with the same mean and variance

gamm<-function (n, a, b)
{
mu <- a/b
sig <- sqrt(a/(b * b))
```

```
vec <- vector("numeric", n)
x <- a/b
vec[1] <- x
for (i in 2:n) {
    can <- rnorm(1, mu, sig)
    importance.ratio<-(dgamma(can, a, b)/dgamma(x,a, b))
    adjustment<-(dnorm(can, mu, sig)/dnorm(x,mu,sig))
    aprob <- min(1,importance.ratio/adjustment)
    u <- runif(1)
    if (u < aprob)
    x <- can
    vec[i] <- x
}
vec
}

vec<-gamm(10000,2.3,2.7)
op<-par(mfrow=c(2,1))
plot(ts(vec))
hist(vec,30)
```





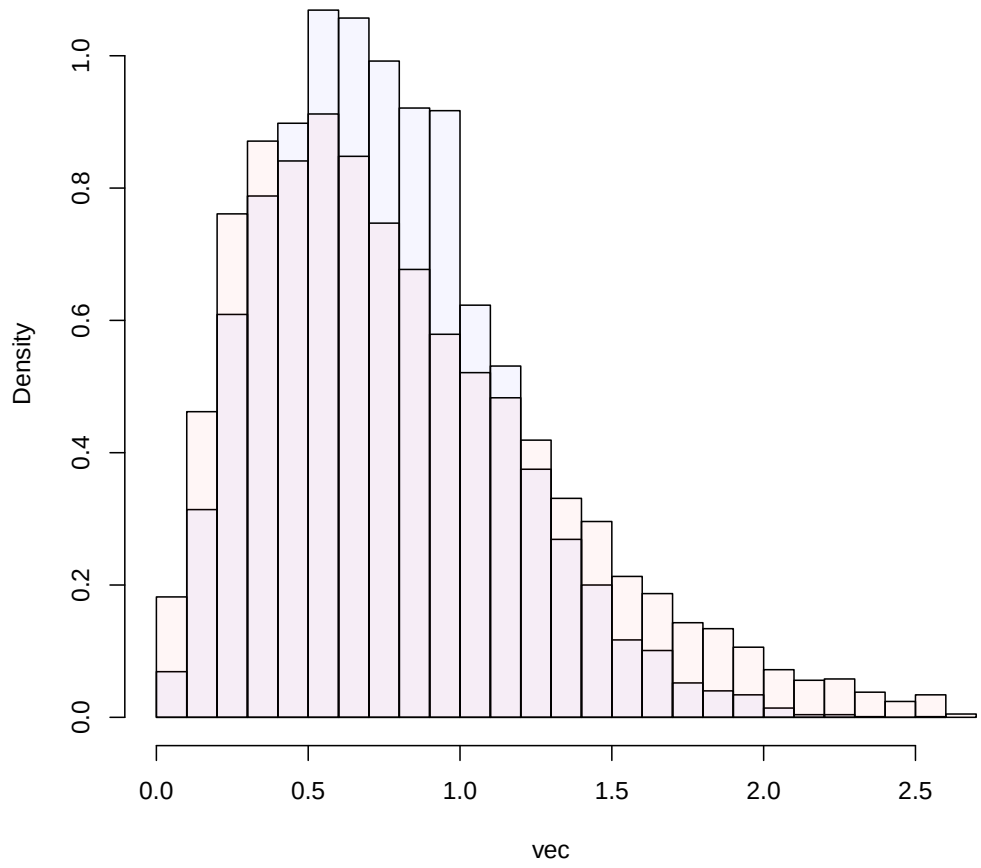

```
# end
vec.orig<-vec
```

We can see why we need the adjustment in the ratio R by simply removing it. Without the adjustment we get a mis-characterization of the posterior distribution of interest.

```
gamm<-function (n, a, b)
{
mu <- a/b
```

```
sig <- sqrt(a/(b * b))
vec <- vector("numeric", n)
x <- a/b
vec[1] <- x
for (i in 2:n) {
    can <- rnorm(1, mu, sig)
    importance.ratio<-(dgamma(can, a, b)/dgamma(x,a, b))
    adjustment<-(dnorm(can, mu, sig)/dnorm(x,mu,sig))
    #aprob <- min(1,importance.ratio/adjustment)
    aprob <- min(1,importance.ratio)
    u <- runif(1)
    if (u < aprob)
    x <- can
    vec[i] <- x
}
vec
}

vec<-gamm(10000,2.3,2.7)
```

```
hist(vec,30,col=addTrans(rep("blue",length(vec)),10),freq=F)
hist(vec.orig,30,col=addTrans(rep("red",length(vec.orig)),10),
     freq=F,add=T)

## Warning in if (!add) {: the condition has length > 1 and only
the first element will be used
```



Note that we have not explained why the Metropolis-Hastings algorithm works.An intuitive explanation is that once a sample from the target distribution has been obtained, all subsequent samples will be from the target distribution. There are more details in Gilks et al book, and the Handbook of Markov Chain Monte Carlo.

12 Assessing convergence: Fat hairy caterpillars

Visually, you can assess convergence by looking at the chain to see if it looks like a "fat hairy caterpillar" (This is how Lunn et al. describe it.)

If you use multiple chains with different starting values, and pool all the chains to assess convergence, you will generally get convergence (at least in the models we consider in this course). In our linear mixed models (coming up) we use 3 or 4 chains. The downside of running multiple chains is computational overhead. However, for regular psycholinguistic data this is not going to be a major problem. (It will probably be a major problem for ERP data; but we'll just throw computational power at the problem by simply using a more powerful computing environment than a laptop).

The Gelman-Rubin (or Brooks-Gelman-Rubin) diagnostic involves sampling from multiple chains with "overdispersed" original values, and then comparing between and within group variability. Within variance is represented by the mean width of the 95% posterior Credible Intervals (CrI) of all chains, from the final T iterations. Within variance is represented by the width of the 95% CrI using all chains pooled together (for the T iterations). If the ratio $\hat{R} = B/W$ is approximately 1, we have convergence. Alternatively, you can run the model on incrementally increasing blocks of iterations T_i, e.g., $T_1 = 1000, T_2 = 2000$, etc., and assess the emergence of convergence.

13 Other sampling methods

There are other methods of sampling available than the ones presented above. Examples are slice sampling and adaptive rejection sampling. But for our purposes we only need to know how the Gibbs and MH algorithms work, and that too only *in principle*; we will never need to implement them for data analysis. Stan will do the sampling for us (using HMC of course).

References

[1] Steve Brooks, Andrew Gelman, Galin Jones, and Xiao-Li Meng. *Handbook of Markov chain Monte Carlo*. CRC Press, 2011.

[2] Walter R Gilks, Sylvia Richardson, and David J Spiegelhalter. *Markov chain Monte Carlo in practice*, volume 2. CRC press, 1996.

[3] Scott Michael Lynch. *Introduction to applied Bayesian statistics and estimation for social scientists*. Springer, 2007.