Bayes: Homework 6
Leslie Gains-Germain

1. (a) I show my code for the Metropolis algorithm below. I start by writing a function for the jumping distribution and the log posterior distribution. I used a standard deviation of 1 for the jumping distribution. On the next page, I show the process I went through in selecting this standard deviation.

```
#1. Specify a symmetric jumping/proposal distribution
# Uniform distribution centered the current value of theta and scaled by sd.scale

  draw.cand.fun <- function(cur, sd.scale) {
       theta.draw <- runif(1, cur - sqrt(12*sd.scale)/2, sqrt(12*sd.scale)/2 + cur)
       return(theta.draw)
  }
  #draw.cand.fun(6, sd.scale=1)  #check the function
```

```
#2. Write a function to evaluate the posterior distribution
#   at theta for a given vector of responses
  ## do on log scale

  logpost.fun <- function(theta,  y.vec) {
     out <- log(prod(1/(1+(y.vec-theta)^2))*1/sqrt(2*pi*5)*exp(-(theta-20)^2/10))
     return(out)
    }
#logpost.fun(1, c(23, 25, 26)) #check function
```

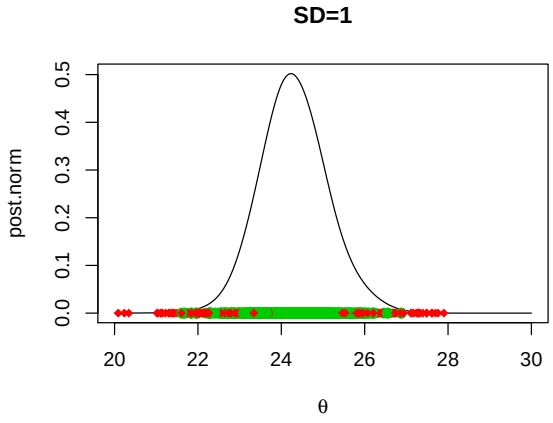
```
#2. Get a starting value for theta

    theta.start <- 24

#3. Define a few things before we start

    y.obs <- c(23, 24, 25, 26.5, 27.5)  #observed data

   nsim <-1000  #number of iteration
   theta.vec <- rep(NA, nsim) #empty vector for theta values
   r.vals <- numeric(nsim-1)  #keep track of r's
   jump.vec <- numeric(nsim-1) #keep track of when we jump (accept candidates)
   theta.vec[1] <- theta.start
```

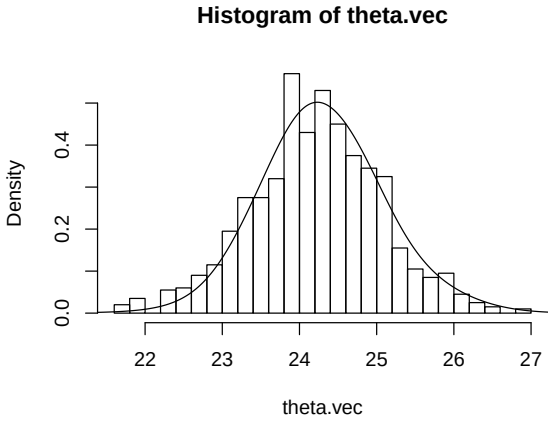
```
#Run Metropolis Algorithm
 for (i in 2:nsim) {
   theta.cur <- theta.vec[i-1]
   theta.cand <- draw.cand.fun(theta.cur, sd.scale=1)
    points(theta.cand, 0, pch=18, cex=1, col=2)
```

```
log.post.cur <- logpost.fun(theta.cur, y.vec=y.obs)
log.post.cand <- logpost.fun(theta.cand, y.vec=y.obs)
log.r <- log.post.cand - log.post.cur

p.accept <- min(1, exp(log.r))

u <- runif(1)
ifelse(u <= p.accept, theta.vec[i]<- theta.cand, theta.vec[i] <- theta.cur)

points(theta.vec[i], 0, pch=16, cex=1.2, col=3)
 lines(c(theta.vec[i-1], 0), c(theta.vec[i], 0), col=4)

jump.vec[i-1] <- ifelse(u <= p.accept, 1, 0)
r.vals[i-1] <- round(exp(log.r), digits=3)
}
```

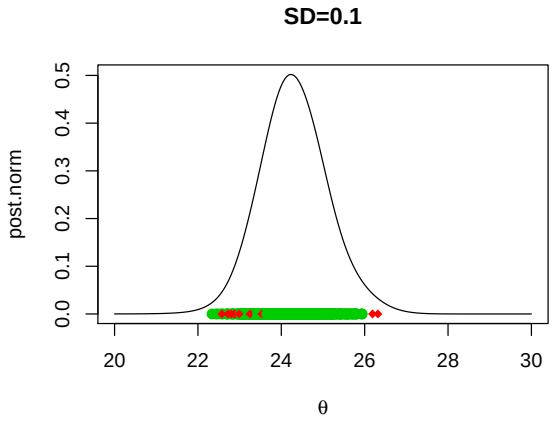
The normalized posterior distribution is plotted below. The right plot shows the normalized posterior distribution found using a grid approximation with the histogram of draws obtained via the Metropolis algorithm. The left plot shows the normalized posterior distribution with green dots for candidate values that were accepted and red dots for candidate values that were not accepted. The ratio of green dots to red dots seems appropriate, indicating that a standard deviation of 1 was a good choice for the jumping distribution. Additionally, the proportion of candidate values that are accepted is right around 0.6, which is a good acceptance ratio. In the next part, I show what the results looked like in my exploration of different tuning parameters.

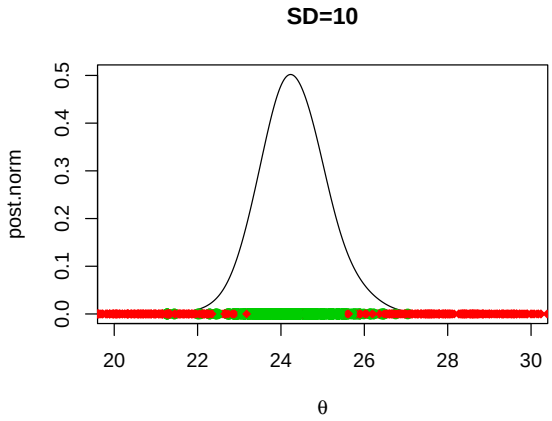
A 99% posterior interval for θ found from the draws of the Metropolis Algorithm is $(21.91, 26.15)$. The posterior probability that θ is greater than 30 is less than 0.0001. The posterior probability that θ is greater than 25 is 0.206.





In the left hand plot below, I used a standard deviation of 0.1 for the jumping distribution. We can see that many more candidate values were accepted than rejected, and the range of candidate values was relatively narrow. It bothers me that we didn't have the opportunity to sample from a wider range of values in the parameter space. The proportion of acceptance was 0.88 for the standard deviation of 0.1. In the right hand plot, I used a standard deviation of 10 for the jumping distribution. There are many more candidate values that were rejected, and the proportion of acceptance was 0.25. Here, we did get the opportunity to sample from a wider range of parameter values, but the algorithm was very inefficient because so many draws were rejected. The standard deviation of 1, shown in the plot above, is a nice compromise between these two extremes.

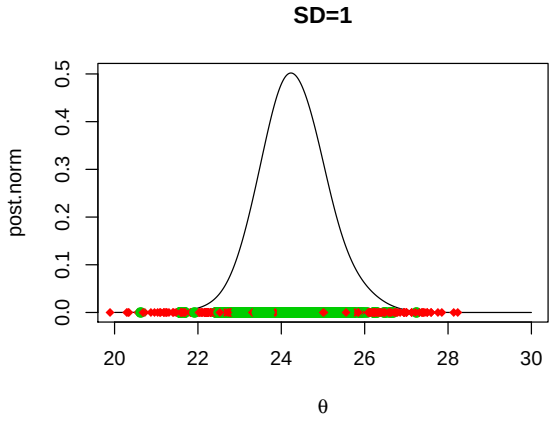


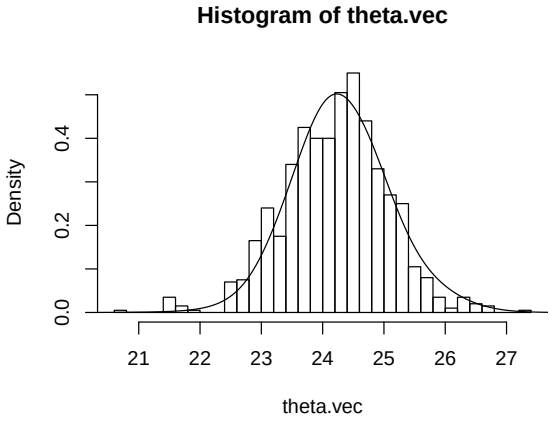


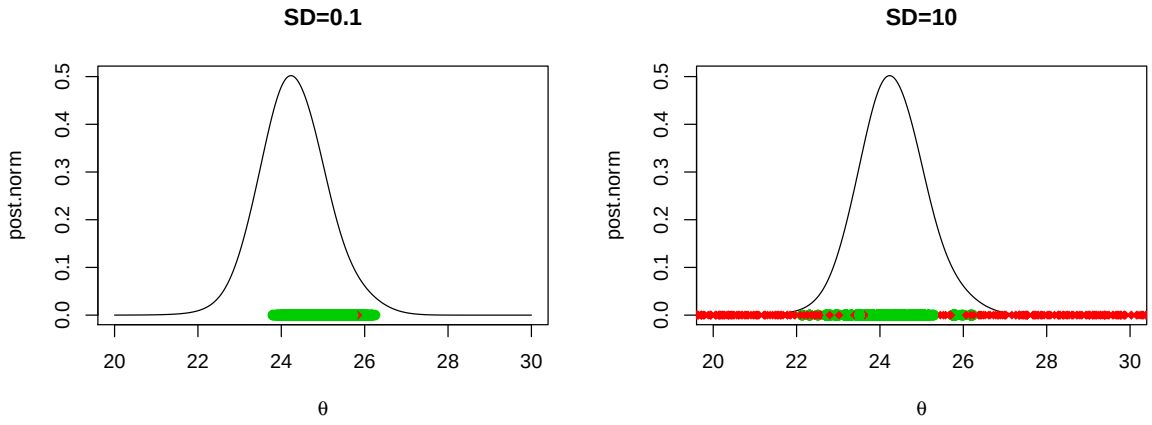
(b) I re-ran the Metropolis algorithm with a normal jumping distribution with a standard

deviation of 1, and the histogram of draws is shown below. With a standard deviation of 1, the proportion of draws accepted was 0.627. I then experimented, again, with different standard deviations. With a standard deviation of 0.1, the proportion of draws accepted was 0.968, and with a standard deviation of 10, the proportion of draws accepted was 0.090. Again, we see that many draws are rejected when the standard deviation is too large, and when the standard deviation is too small, it takes longer (more simulations) to obtain a sample from the entire parameter space.

A 99% posterior interval for θ found from the draws obtained via the Metropolis algorithm with the normal jumping distribution is $(22.44, 26.28)$. The posterior probability that θ is greater than 30 is still less than 0.0001, and the posterior probability that θ is greater than 25 is 0.175.







(c) The results were similar for both jumping distributions. The posterior intervals and probabilities for both jumping distributions are shown in the table below.

	Uniform Jumping Distribution	Normal Jumping Distribution
99% posterior interval	$(21.91, 26.15)$	$(22.44, 26.28)$
$Pr(> 25)$	0.206	0.175
acceptance ratio SD=1	0.61	0.627
acceptance ratio SD=0.1	0.88	0.968
acceptance ratio SD=10	0.25	0.090

The uniform jumping distribution was easier to tune. It seems like the effect of increasing or decreasing the standard deviation is more extreme when using the normal jumping distribution. The normal jumping distribution acceptance ratios, shown in the table below, at standard deviations of 0.1 and 10 were worse (farther from 0.5) than the uniform jumping distribution acceptance ratios at these standard deviations. The message to take away is that changing the standard deviation of the normal jumping distribution will have a larger effect on the acceptance ratio compared to the same change in the standard deviation of the uniform jumping distribution.

I think I prefer the uniform jumping distribution because it is more convenient to use the range as the tuning parameter for the uniform jumping distribution rather than

the standard deviation. In this example, I did use the standard deviation as the tuning parameter for comparison with the normal, but in the future it would be easier to specify the range as the tuning parameter when using a uniform jumping distribution. I really like using range as a measure of spread because it is easy for me to think about the meaning of the range of a uniform distribution and how it relates to spread. Since the results are similar from the two jumping distributions, I think it makes the most sense to choose the jumping distribution that is easiest for me to visualize.

2. Below is the code for my Metropolis-Hastings algorithm used to obtain draws from a $Poi(8)$ distribution using a Negative-Binomial proposal distribution.

```
#1. Specify a symmetric jumping/proposal distribution
# Negative Binomial distribution w/ mean the current value of y and tuning param sd.scale

  draw.cand.fun3 <- function(cur, sd.scale) {
       y.draw <- rnbinom(1, size=sd.scale, mu=(cur+0.1))
       return(y.draw)
  }
  #draw.cand.fun3(2, 1)  #check the function
```

```
logjump.fun <- function(value, mean, sd.scale){
  out <- log(dnbinom(value, size=sd.scale, mu=(mean+0.1)))
  return(out)
}
```

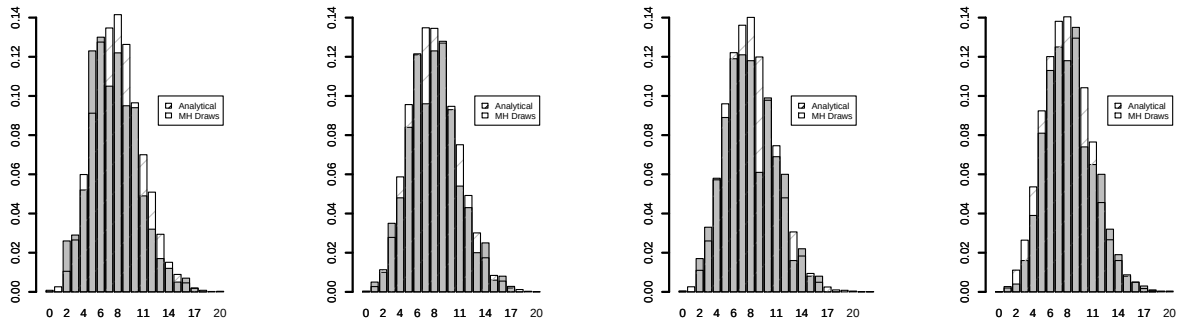
```
#2. Write a function to evaluate the target distribution
#   at y for a given lambda

  logtarget.fun <- function(lambda, y) {
     out <- log(exp(-lambda)*lambda^y/factorial(y))
     return(out)
    }

#target.fun(8,3) check function
```

```
#2. Get a starting value for mu

    y.start <- 7

#3. Define a few things before we start
```

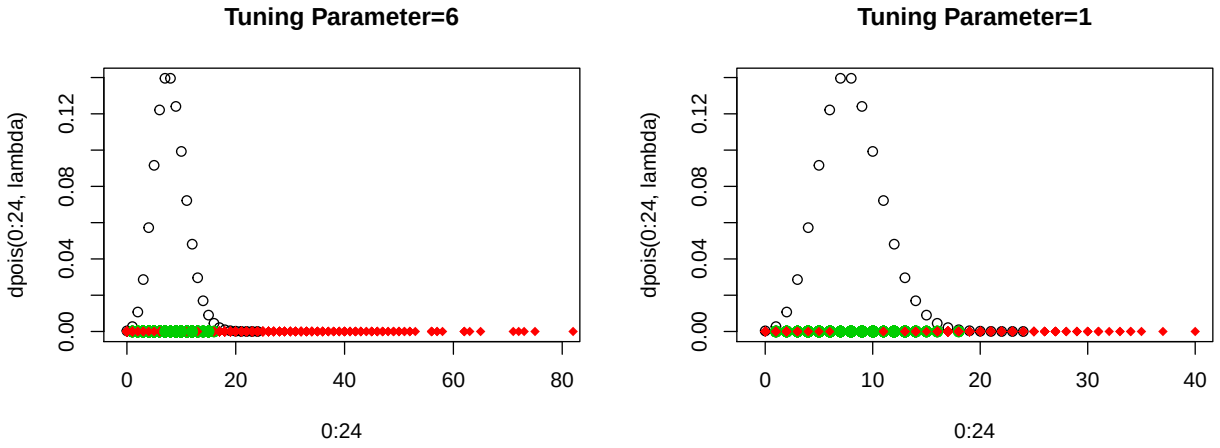
```
   nsim <- 1000  #number of iterations
   y.mat <- matrix(NA, nrow=nsim, ncol=4) #empty matrix for y draws
   lambda <- 8
   r.vals <- matrix(NA, nrow = nsim-1, ncol = 4)
   jump.vec <- matrix(NA, nrow = nsim-1, ncol = 4)
   y.mat[1,] <- y.start

   sd.scale <- 6
```

```
for(j in 1:4){
 for (i in 2:nsim) {
   y.cur <- y.mat[i-1, j]
   y.cand <- draw.cand.fun3(y.cur, sd.scale)

   logtarget.cur <- logtarget.fun(lambda, y.cur)
   logtarget.cand <- logtarget.fun(lambda, y.cand)

   logjump.cur <- logjump.fun(value=y.cur, mean=y.cand, sd.scale)
   logjump.cand <- logjump.fun(value=y.cand, mean=y.cur, sd.scale)

   log.r <- logtarget.cand - logtarget.cur + logjump.cur - logjump.cand

   p.accept <- min(1, exp(log.r))

   u <- runif(1)
   ifelse(u <= p.accept, y.mat[i, j] <- y.cand, y.mat[i, j] <- y.cur)

   jump.vec[i-1, j] <- ifelse(u <= p.accept, 1, 0)
   r.vals[i-1, j] <- round(exp(log.r), digits=3)
   }
}
```

I ran four chains, and below are the plots of 900 draws obtained from my Metropolis Hastings algorithm with the analytical Poisson distribution overlaid (for each chain). The analytical distribution is represented by the unshaded bars. I dropped the first 100 draws because I assume that it took 100 iterations for the algorithm to converge.



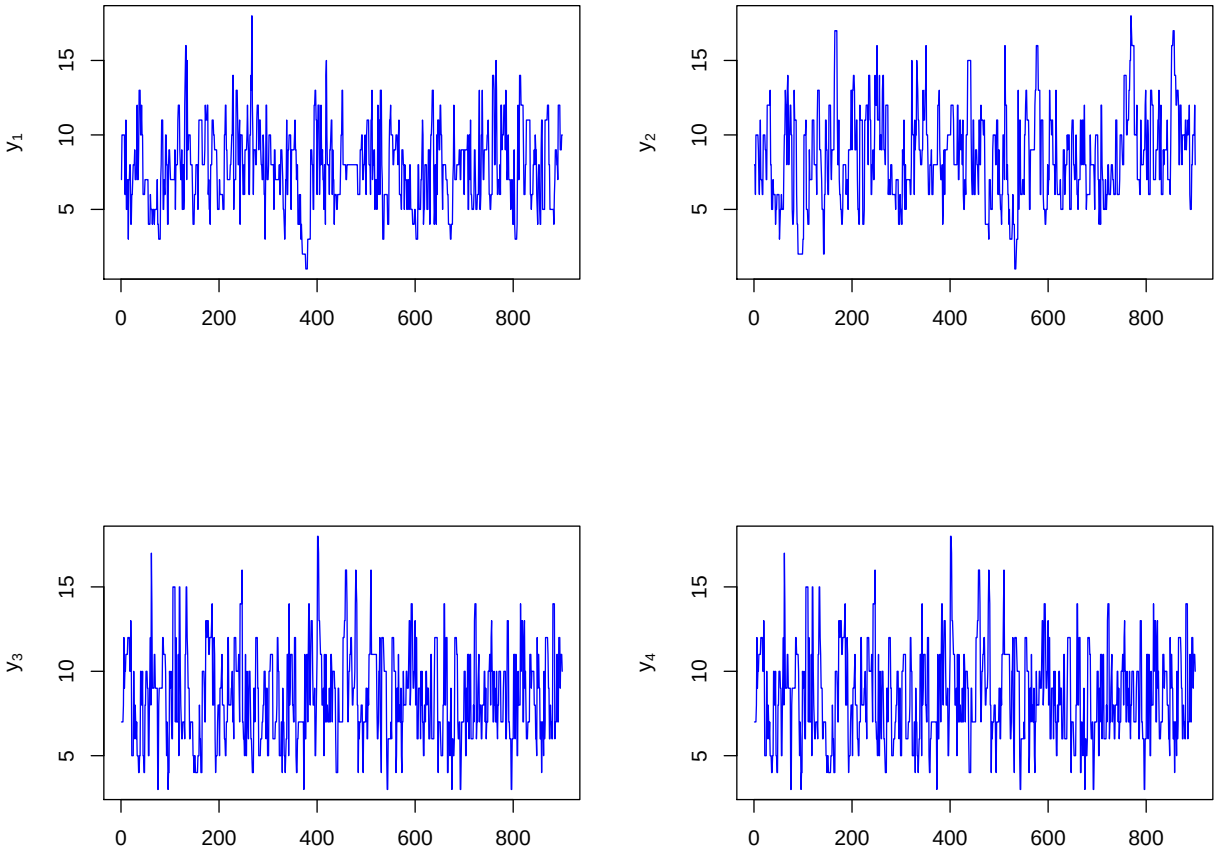
To tune the algorithm, I looked at the acceptance ratio for different values of `sd.scale`, and I looked at plots that gave me a visual of the proportion of rejected candidate draws and the values of the candidates. In the plot on the left below, the tuning parameter was set to 1, and I can clearly see that the proportion of rejected candidate draws is too high. Additionally, the values of the candidate draws went up to 80, which is way too high. In the plot on the right below, the tuning parameter was set to 6. Fewer candidate draws were rejected, and the values of the draws were more reasonable. I decided on a tuning parameter of 6 because the acceptance ratios were between 0.5 and 0.6 for all four chains.




```
apply(jump.vec, 2, mean)
```

```
## [1] 0.575 0.553 0.545 0.592
```

Lastly, I assessed convergence by making sample path plots that show the draws for each iteration of the algorithm. The burn in period of the first 100 iterations is not shown. For each chain, we appear to get draws from the entire range of possible values (about 3 to 15). Also, the draws do not appear to be stuck for too long in one area before moving to another area of the space. The fact that all four of these plots look like random noise indicates that the algorithm converged.

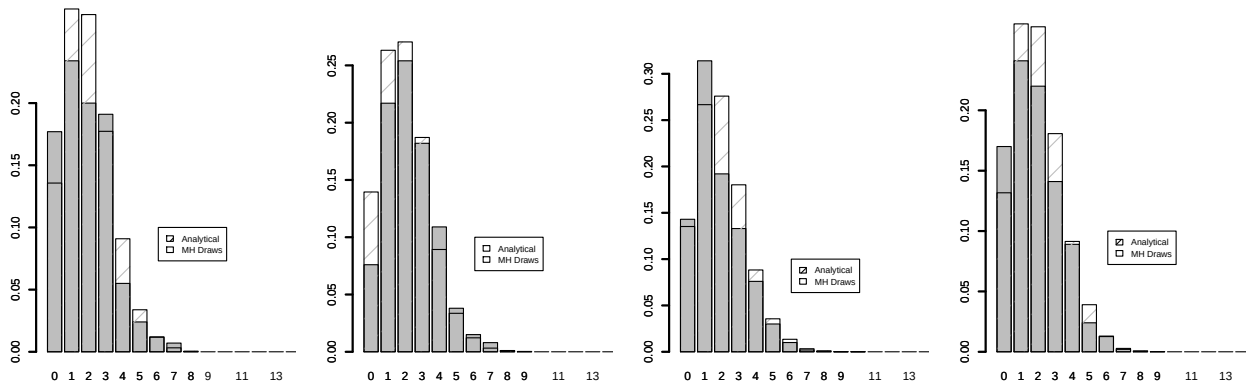


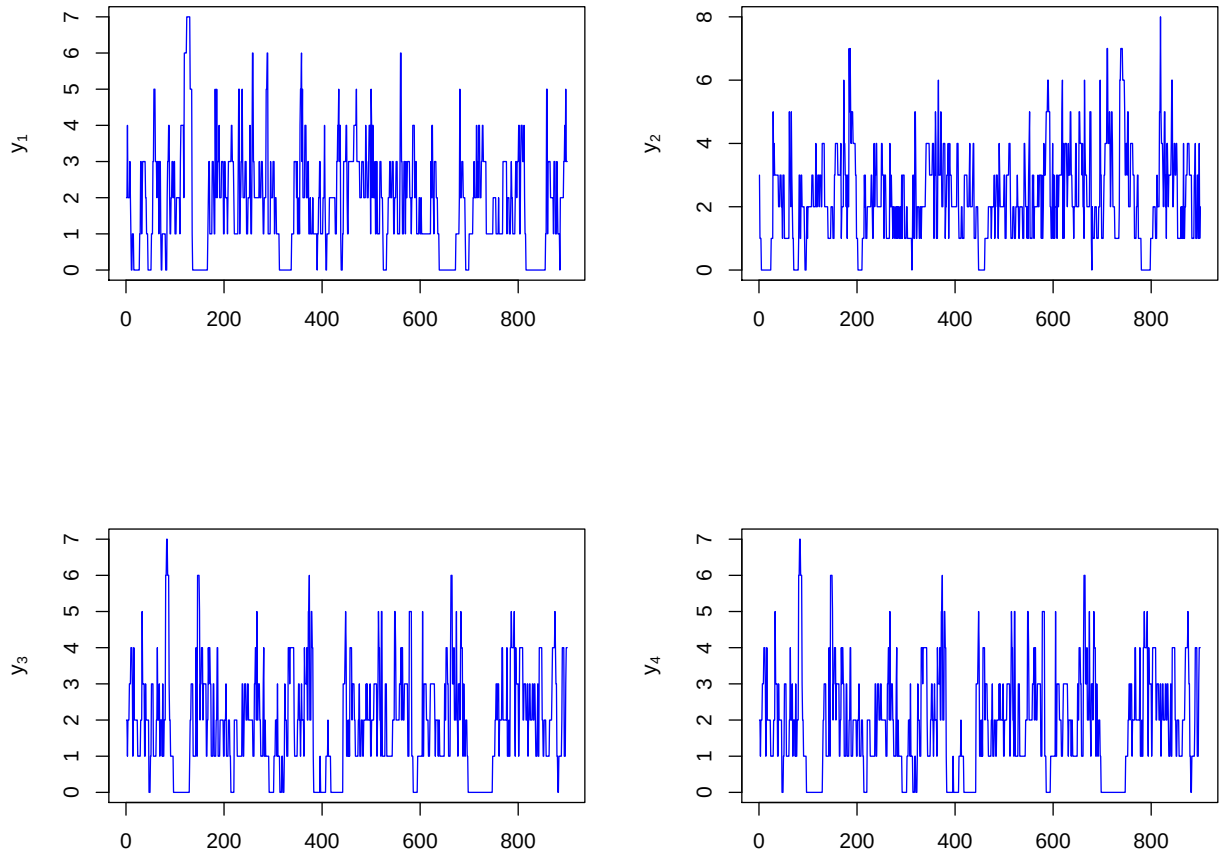
I did fool around with writing a Metropolis-Hastings algorithm to draw from a $Poi(2)$ distribution. Instead of using a negative binomial jumping distribution with mean equal to the current value of y, I made the mean equal to the current value of y plus 0.1. I have to add 0.1 so that the algorithm won't get stuck at 0 (the only possible value that can be drawn from a negative binomial distribution with a mean of 0 is 0 itself). This strategy works OK, but you

can still see that it takes the algorithm a while to move away from 0. This is why there are long lines at 0 in every one of the sample path plots. I think it would actually be better to make the mean of the proposal distribution y plus 1. This way, if 0 is accepted, the next draw will be taken from a negative binomial distribution with a mean of 1 (rather than a mean of 0.1).

```
#1. Specify a symmetric jumping/proposal distribution
# Negative Binomial distribution w/ mean the current value of y and tuning param sd.scale

  draw.cand.fun3 <- function(cur, sd.scale) {
       y.draw <- rnbinom(1, size=sd.scale, mu=(cur+0.1))
       return(y.draw)
  }
  #draw.cand.fun3(2, 1)  #check the function
```

```
logjump.fun <- function(value, mean, sd.scale){
  out <- log(dnbinom(value, size=sd.scale, mu=(mean+0.1)))
  return(out)
}
```





3. When using Bayesian methods, we often want to describe a posterior distribution for a parameter of interest. Sometimes, the posterior is a known distribution such as a Normal, Beta, or Gamma distribution. If this is the case, it's easy to describe the posterior because these distributions have already been described by others! It is not often the case, however, that the posterior is a known distribution. Often, the posterior is unrecognizable, and we can only solve analytically for a scalar multiple of the posterior density function. The way that we describe these strange posterior distributions is by taking lots of random draws from them. With a large number of random draws from a posterior distribution, we can provide summary measures for the distribution (mean, median, variance, etc) as well as posterior intervals and posterior probabilities. MCMC methods are used to take random draws from these wacky,

unrecognizable posterior functions. Although MCMC methods are used to generate draws from posterior distributions in Bayesian Statistics, these methods can be used to randomly sample from any distribution of interest in any discipline.

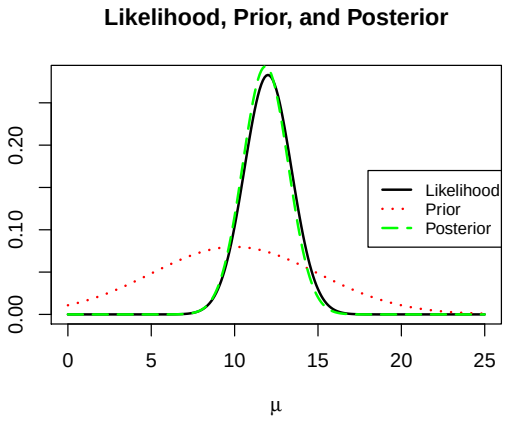
4. (a) Suppose $y|\mu \sim N(\mu, 2)$, and $\mu \sim N(\mu_0, \sigma_0^2)$. Then $\mu|y \sim N(\mu_1, \sigma_1^2)$ where μ_1 and σ_1^2 are given as follows. This result was shown in class.

$$\mu_1 = \frac{1/\sigma_0^2 * \mu_0 + 1/2 * (12)}{1/\sigma_0^2 + 1/2}$$
$$\sigma_1^2 = \frac{2\sigma_0^2}{\sigma_0^2 + 2}$$

So, the analytical posterior distribution is:

$$p(\mu|y) = \frac{1}{\sqrt{2\pi\sigma_1^2}} e^{-(\mu-\mu_1)/2\sigma_1^2}$$

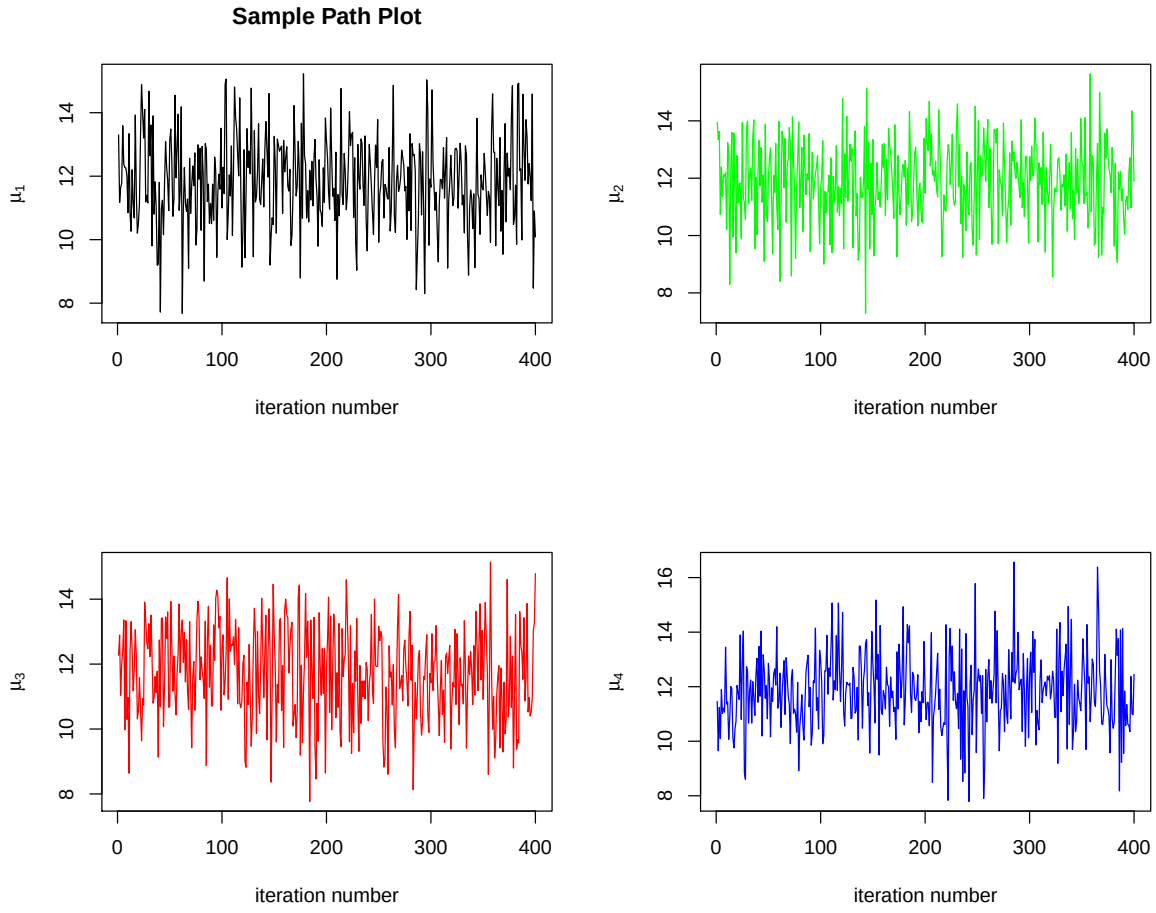
I will assume that $\mu_0 = 10$ and $\sigma_0^2 = 25$. Solving for μ_1 and σ_1^2, we find that $\mu_1 = 320/27 \approx 11.85$, and $\sigma_1^2 = 50/27 \approx 1.85$. Plugging in these values, the posterior distribution is $N(320/27, 50/27)$. The plot of the likelihood, prior, and posterior distribution is shown below.



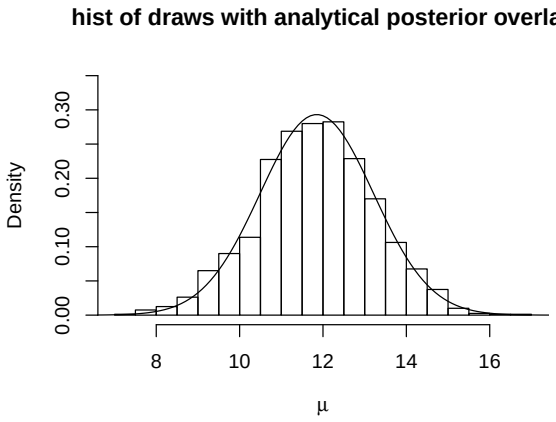
(b) My JAGs model code is below.

```
setwd("~/Documents/Stat532/homeworks/bayeshw6")

##write model file first
cat("
model
{
y ~ dnorm(mu, 1/2)
mu~dnorm(mu0, tau0)
tau0 <-pow(sigma0, -2)
mu0 <- 10
sigma0 <- 5
}",
file="jags-intro.jags")
```

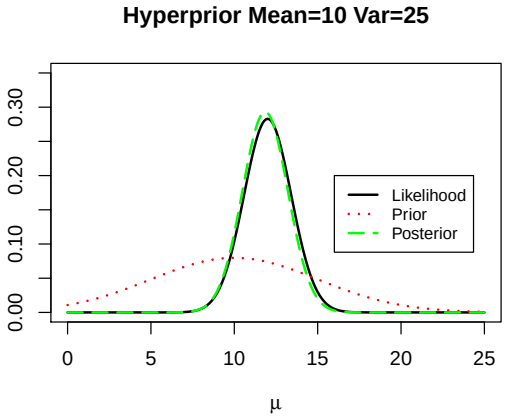
(c) The sample path plots are shown below. 500 iterations were run, but the burn-in period of the first 100 iterations is not shown. 4 chains were run.

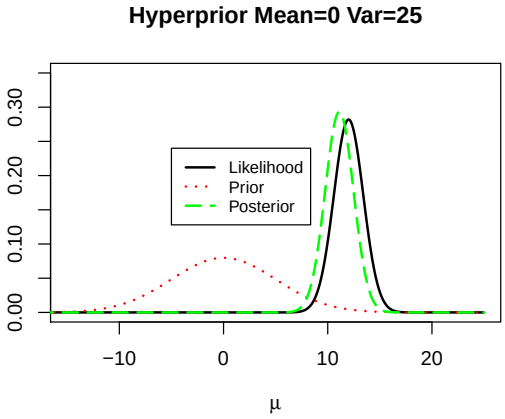


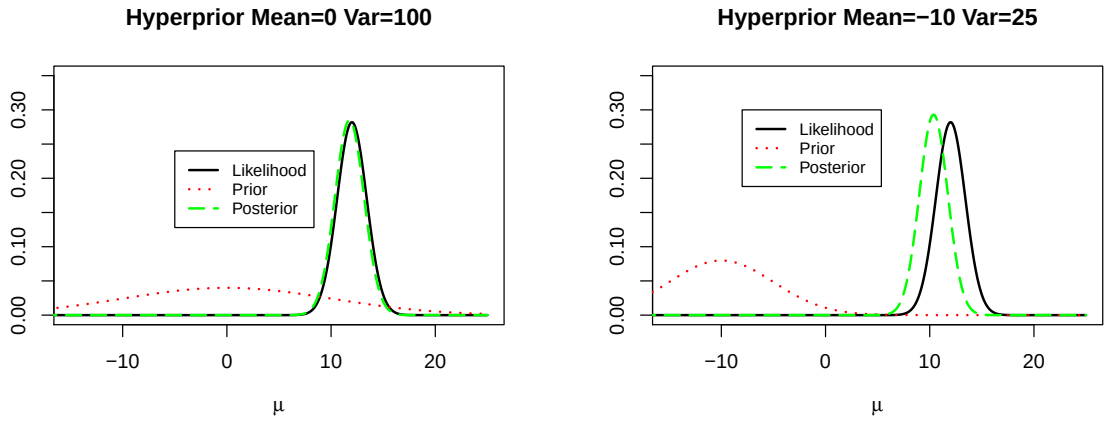
(d) Below is the histogram of all posterior draws with a curve of analytical posterior density function overlaid on it.



(e) The plot of the likelihood, prior, and posterior distribution is shown below for different values of the hyperparameters. I experimented by changing both the mean and the variance of the prior. I noticed that as the prior variance increased, the posterior mean got closer to the observed data value of 12. When the prior had smaller variance, the posterior mean was closer to the prior mean. The other thing I noticed was that when the variance of the prior was fixed at 25, the farther the prior mean was from 12, the farther the posterior mean was from 12. With a really small sample size of 1, I would expect to see noticeable changes in the posterior distribution as the prior parameters change. I imagine that the effect of the prior parameters decreases as the sample size increases.







(f) JAGS: Just Another Gibbs Sampler

BUGS: Bayesian Inference Using Gibbs Sampling

STAN: The Gelman book says it stands for Sampling Through Adaptive Neighborhoods, but I also found online that it stands for Stanislaw Ulam, the Polish-American mathematician who first proposed the Monte Carlo Method

5. (a) The posterior distributions were found computationally "using Markov Chain Monte Carlo algoirthms as implemented in the WinBUGS software platform" (Sethi 9). From looking online, it sounds like WinBUGS is simply a version of BUGs for Windows. WinBUGs uses a GIBBS sampler to obtain draws from the posterior, and it uses a metropolis algorithm to sample from difficult complete conditional distributions. It sounds like WinBUGS and JAGs are exactly the same computationally. WinBUGs came first, and then JAGs was developed to be an open source version that runs on Linux and can interface to R. Now, clearly either WinBUGs or JAGs can interface to R (using R2jags or R2WinBUGs), but originally I think it wasn't as easy to run BUGs in other programming environments. In practice, I think there are slight differences in the way code is written in JAGs compared to BUGs. Again, I'll reiterate that from what I understand the computational algorithm used to obtain draws from the posterior distribution is the same in both programs. Essentially, the authors of my paper are using the canned soft-

ware package WinBUGs to implement a Gibbs Sampler to approximate the posterior distributions of interest.

The goal of this paper is to describe a new way to impute run sizes for missing dates at a fish weir. I think they chose to use WinBUGS software because "WinBUGS treats missing dates as parameters to be estimated and returns a posterior distribution for the passage estimate for each missing date" (Sethi 10). So, they get posterior distributions for the parameters describing the run curves and process variation models, but they also end up with posterior distributions for the run sizes at each missing date. Finally, they can then get a posterior distribution for the total run size by summing all observed passages and all estimated passages for missing dates.

(b) They used $\hat{R}$ to assess convergence. They used a cutoff value of 1.2, stating that the algorithm did not converge if $\hat{R}$ was greater than 1.2. They then went on to assess under what conditions convergence failure occurred. For example, they say that "convergence was most common for simulation scenarios with the initial 15% of the run missing." They don't mention using any other convergence diagnostics other than $\hat{R}$.

I'm curious what your (Megan) opinion is on using $\hat{R}$ cutoff values to assess convergence. On the one hand, the cutoff is arbitrary. On the other hand, it seems like you have to make a decision at some point about whether the draws obtained from the MCMC algorithm can be trusted to approximate the posterior of interest.

R code appendix

```
par(mfrow=c(1,2))

plot(theta.plot[,1], post.norm, type="l", main="SD=0.1", xlab=expression(theta))
points(theta.start, 0, col=2, pch=16, cex=1.2)

#small sd.scale
 for (i in 2:nsim) {
   theta.cur <- theta.vec[i-1]
   theta.cand <- draw.cand.fun(theta.cur, sd.scale=.1)
```

```
    points(theta.cand, 0, pch=18, cex=1, col=2)

   log.post.cur <- logpost.fun(theta.cur, y.vec=y.obs)
   log.post.cand <- logpost.fun(theta.cand, y.vec=y.obs)
   log.r <- log.post.cand - log.post.cur

   p.accept <- min(1, exp(log.r))

   u <- runif(1)
   ifelse(u <= p.accept, theta.vec[i]<- theta.cand, theta.vec[i] <- theta.cur)

   points(theta.vec[i], 0, pch=16, cex=1.2, col=3)
    lines(c(theta.vec[i-1], 0), c(theta.vec[i], 0), col=4)

   jump.vec[i-1] <- ifelse(u <= p.accept, 1, 0)
   r.vals[i-1] <- round(exp(log.r), digits=3)
   }

plot(theta.plot[,1], post.norm, type="l", main="SD=10", xlab=expression(theta))
points(theta.start, 0, col=2, pch=16, cex=1.2)

#large sd.scale
 for (i in 2:nsim) {
   theta.cur <- theta.vec[i-1]
   theta.cand <- draw.cand.fun(theta.cur, sd.scale=10)
    points(theta.cand, 0, pch=18, cex=1, col=2)

   log.post.cur <- logpost.fun(theta.cur, y.vec=y.obs)
   log.post.cand <- logpost.fun(theta.cand, y.vec=y.obs)
   log.r <- log.post.cand - log.post.cur

   p.accept <- min(1, exp(log.r))

   u <- runif(1)
   ifelse(u <= p.accept, theta.vec[i]<- theta.cand, theta.vec[i] <- theta.cur)

   points(theta.vec[i], 0, pch=16, cex=1.2, col=3)
    lines(c(theta.vec[i-1], 0), c(theta.vec[i], 0), col=4)

   jump.vec[i-1] <- ifelse(u <= p.accept, 1, 0)
   r.vals[i-1] <- round(exp(log.r), digits=3)
   }
```

```
#1. Specify a symmetric jumping/proposal distribution
# normal distribution centered the current value of theta and scaled by sd.scale

  draw.cand.fun2 <- function(cur, sd.scale) {
       theta.draw <- rnorm(1, cur, sd=sd.scale)
       return(theta.draw)
  }
  #draw.cand.fun2(6, sd.scale=1)  #check the function
```

```
par(mfrow=c(1,2))

plot(theta.plot[,1], post.norm, type="l", main="SD=0.1", xlab=expression(theta))
points(theta.start, 0, col=2, pch=16, cex=1.2)

#small sd.scale
 for (i in 2:nsim) {
   theta.cur <- theta.vec[i-1]
   theta.cand <- draw.cand.fun2(theta.cur, sd.scale=.1)
    points(theta.cand, 0, pch=18, cex=1, col=2)

   log.post.cur <- logpost.fun(theta.cur, y.vec=y.obs)
   log.post.cand <- logpost.fun(theta.cand, y.vec=y.obs)
   log.r <- log.post.cand - log.post.cur

   p.accept <- min(1, exp(log.r))

   u <- runif(1)
   ifelse(u <= p.accept, theta.vec[i]<- theta.cand, theta.vec[i] <- theta.cur)

   points(theta.vec[i], 0, pch=16, cex=1.2, col=3)
    lines(c(theta.vec[i-1], 0), c(theta.vec[i], 0), col=4)

   jump.vec[i-1] <- ifelse(u <= p.accept, 1, 0)
   r.vals[i-1] <- round(exp(log.r), digits=3)
   }

plot(theta.plot[,1], post.norm, type="l", main="SD=10", xlab=expression(theta))
points(theta.start, 0, col=2, pch=16, cex=1.2)

#large sd.scale
 for (i in 2:nsim) {
   theta.cur <- theta.vec[i-1]
   theta.cand <- draw.cand.fun2(theta.cur, sd.scale=10)
    points(theta.cand, 0, pch=18, cex=1, col=2)

   log.post.cur <- logpost.fun(theta.cur, y.vec=y.obs)
   log.post.cand <- logpost.fun(theta.cand, y.vec=y.obs)
   log.r <- log.post.cand - log.post.cur

   p.accept <- min(1, exp(log.r))

   u <- runif(1)
   ifelse(u <= p.accept, theta.vec[i]<- theta.cand, theta.vec[i] <- theta.cur)

   points(theta.vec[i], 0, pch=16, cex=1.2, col=3)
    lines(c(theta.vec[i-1], 0), c(theta.vec[i], 0), col=4)

   jump.vec[i-1] <- ifelse(u <= p.accept, 1, 0)
   r.vals[i-1] <- round(exp(log.r), digits=3)
   }
```

```r
par(mfrow=c(1,4))

burn.in <- 100

y1 <- factor(y.mat[(burn.in+1):nsim,1], levels=c(0:18))
y2 <- factor(y.mat[(burn.in+1):nsim,2], levels=c(0:18))
y3 <- factor(y.mat[(burn.in+1):nsim,3], levels=c(0:18))
y4 <- factor(y.mat[(burn.in+1):nsim,4], levels=c(0:18))

#plot draws from Metropolis Hastings algorithm with Poisson dist overlaid
barplot(table(y1)/nsim, ylim=c(0, 0.15), xlim=c(0, 30))
barplot(table(rpois(10000, lambda))/10000, add=TRUE, density=5, lwd=2, xlim=c(0, 20))
legend(16, 0.1, c("Analytical", "MH Draws"), density=c(5,0), cex=0.7)

barplot(table(y2)/nsim, ylim=c(0, 0.15), xlim=c(0, 30))
barplot(table(rpois(10000, lambda))/10000, add=TRUE, density=5, lwd=2, xlim=c(0, 20))
legend(16, 0.1, c("Analytical", "MH Draws"), density=c(5,0), cex=0.7)

barplot(table(y3)/nsim, ylim=c(0, 0.15), xlim=c(0, 30))
barplot(table(rpois(10000, lambda))/10000, add=TRUE, density=5, lwd=2, xlim=c(0, 20))
legend(16, 0.1, c("Analytical", "MH Draws"), density=c(5,0), cex=0.7)

barplot(table(y4)/nsim, ylim=c(0, 0.15), xlim=c(0,30))
barplot(table(rpois(10000, lambda))/10000, add=TRUE, density=5, lwd=2, xlim=c(0, 20))
legend(16, 0.1, c("Analytical", "MH Draws"), density=c(5,0), cex=0.7)
```

```r
  burn.in <- 100  #Drop the first 100, b/c hasn't converged yet!
  par(mfrow=c(1,2))
  plot(seq((burn.in+1):nsim), y.mat[(burn.in+1):nsim,1], type="l", ylab=expression(y[1]), col=4, xlab="")
  plot(seq((burn.in+1):nsim), y.mat[(burn.in+1):nsim,2], type="l", ylab=expression(y[2]), col=4, xlab="")
plot(seq((burn.in+1):nsim), y.mat[(burn.in+1):nsim,4], type="l", ylab=expression(y[3]), col=4, xlab="")
plot(seq((burn.in+1):nsim), y.mat[(burn.in+1):nsim,4], type="l", ylab=expression(y[4]), col=4, xlab="")
```

```r
sigma0 <- 5
mu0 <- 10

mu1.fun <- function(mu0, sigma0){
  (mu0/sigma0^2+6)/(1/sigma0^2+1/2)
}
mu1 <- mu1.fun(mu0, sigma0)

sigma1.fun <- function(sigma0){
  2*sigma0^2/(sigma0^2+2)
}
sigma1sq <- sigma1.fun(sigma0)

like.fun <- function(mu){
  1/sqrt(2*pi*2)*exp(-(12-mu)^2/4)
}
```

```
#mu0=10 and sigma0^2=25 (defined above)
prior.fun <- function(mu){
  1/sqrt(2*pi*sigma0^2)*exp(-(mu-mu0)^2/(2*sigma0^2))
}

#then mu1=320/27 and sigma1^2 = 50/27
post.fun <- function(mu){
  1/sqrt(2*pi*(sigma1sq))*exp(-(mu-(mu1))^2/(2*(sigma1sq)))
}

mu <- seq(0, 25, by=0.1)
step <- mu[2]-mu[1]
num.int <- sum(step*like.fun(mu[-151]))
plot(mu, like.fun(mu)/num.int, type="l", xlab=expression(mu), ylab="", main="Likelihood, Prior, and Posterior"
lines(mu, prior.fun(mu), col="red", lty=3, lwd=2)
lines(mu, post.fun(mu), col="green", lty=5, lwd=2)
legend(18, 0.17, c("Likelihood", "Prior", "Posterior"), lty=c(1,3,5), col=c("black", "red", "green"), lwd=c(2,
```

```
##jags call
library(R2jags)
set.seed(52)

data <- list(y=12)

intro.JAGs <-jags(model.file="jags-intro.jags", data = data, parameters.to.save = c("mu"), n.chains=4, n.burni
```

```
par(mfrow=c(1,2))
plot(1:400, as.mcmc(intro.JAGs)[[1]][,2], type="l", xlab="iteration number", ylab=expression(mu[1]), main="Sam
plot(1:400, as.mcmc(intro.JAGs)[[2]][,2], col="green", type="l", xlab="iteration number", ylab=expression(mu[2
plot(1:400, as.mcmc(intro.JAGs)[[3]][,2], col="red", type="l", xlab="iteration number", ylab=expression(mu[3])
plot(1:400, as.mcmc(intro.JAGs)[[4]][,2], col="blue", type="l", xlab="iteration number", ylab=expression(mu[4]
```

```
hist(c(as.mcmc(intro.JAGs)[[1]][,2], as.mcmc(intro.JAGs)[[2]][,2], as.mcmc(intro.JAGs)[[3]][,2], as.mcmc(intro
     xlab=expression(mu), main="hist of draws with analytical posterior overlay")
lines(mu, post.fun(mu))
```

```
par(mfrow=c(1,2))

plot(mu, like.fun(mu)/num.int, type="l", xlab=expression(mu), ylab="", main="Hyperprior Mean=10 Var=25", lwd=2
lines(mu, prior.fun(mu), col="red", lty=3, lwd=2)
lines(mu, post.fun(mu), col="green", lty=5, lwd=2)
legend(16, 0.20, c("Likelihood", "Prior", "Posterior"), lty=c(1,3,5), col=c("black", "red", "green"), lwd=c(2,

sigma0 <- 5
mu0 <- 0

mu1 <- mu1.fun(mu0, sigma0)
```

```
sigma1sq <- sigma1.fun(sigma0)

mu <- seq(-25, 25, by=0.1)
step <- mu[2]-mu[1]
num.int <- sum(step*like.fun(mu[-151]))
plot(mu, like.fun(mu)/num.int, type="l", xlab=expression(mu), ylab="", main="Hyperprior Mean=0 Var=25", lwd=2,
lines(mu, prior.fun(mu), col="red", lty=3, lwd=2)
lines(mu, post.fun(mu), col="green", lty=5, lwd=2)
legend(-5, 0.24, c("Likelihood", "Prior", "Posterior"), lty=c(1,3,5), col=c("black", "red", "green"), lwd=c(2,

sigma0 <- 10
mu0 <- 0

mu1 <- mu1.fun(mu0, sigma0)

sigma1sq <- sigma1.fun(sigma0)

plot(mu, like.fun(mu)/num.int, type="l", xlab=expression(mu), ylab="", main="Hyperprior Mean=0 Var=100", lwd=2
lines(mu, prior.fun(mu), col="red", lty=3, lwd=2)
lines(mu, post.fun(mu), col="green", lty=5, lwd=2)
legend(-5, 0.24, c("Likelihood", "Prior", "Posterior"), lty=c(1,3,5), col=c("black", "red", "green"), lwd=c(2,

sigma0 <- 5
mu0 <- -10

mu1 <- mu1.fun(mu0, sigma0)

sigma1sq <- sigma1.fun(sigma0)

plot(mu, like.fun(mu)/num.int, type="l", xlab=expression(mu), ylab="", main="Hyperprior Mean=-10 Var=25", lwd=
lines(mu, prior.fun(mu), col="red", lty=3, lwd=2)
lines(mu, post.fun(mu), col="green", lty=5, lwd=2)
legend(-8, 0.30, c("Likelihood", "Prior", "Posterior"), lty=c(1,3,5), col=c("black", "red", "green"), lwd=c(2,
```