Week 10 Module

Markov Chain Monte Carlo (MCMC)

In the previous section, we saw that a beta distribution was a conjugate prior for a sampling model, meaning that the posterior was also a beta distribution. This prior specification allows easy posterior computations because the integration in the denominator of Bayes rule $\int p(y|\theta)p(\theta)d\theta$ can be analytically computed.

$$\begin{aligned}
p(\theta|z,N) &= \frac{p(z,N|\theta)p(\theta)}{p(z,N)} \\
&= \frac{p(z,N|\theta)p(\theta)}{\int p(z,N|\theta)p(\theta)d\theta} \\
&= \frac{\theta^z(1-\theta)^{(N-z)} \times (\Gamma(a)\Gamma(b)/\Gamma(a+b))\theta^{(a-1)}(1-\theta)^{(b-1)}}{\int \theta^z(1-\theta)^{(N-z)} \times (\Gamma(a)\Gamma(b)/\Gamma(a+b))\theta^{(a-1)}(1-\theta)^{(b-1)}d\theta} \\
&= \frac{(\Gamma(a)\Gamma(b)/\Gamma(a+b))\theta^{z+a-1}(1-\theta)^{b+N-z-1}}{(\Gamma(a)\Gamma(b)/\Gamma(a+b))\int \theta^{z+a-1}(1-\theta)^{b+N-z-1}d\theta} \\
&= \frac{\theta^{z+a-1}(1-\theta)^{b+N-z-1}}{\int \theta^{z+a-1}(1-\theta)^{b+N-z-1}d\theta} \\
&= \frac{\theta^{z+a-1}(1-\theta)^{b+N-z-1}}{\Gamma(a+b+N)/(\Gamma(a+z)\Gamma(N-z+b))} \\
&= \frac{(\Gamma(a+z)\Gamma(N-z+b))\theta^{z+a-1}(1-\theta)^{b+N-z-1}}{\Gamma(a+b+N)} \\
&\sim Beta(a+z, b+N-z)
\end{aligned}$$

In many situations, this type of prior is not available and we need to use other means to understand the posterior distribution $p(\theta|y)$. MCMC is a tool for taking samples from the posterior when $\int p(y|\theta)p(\theta)d\theta$ is messy and $p(\theta|y)$ does not have the form of a known distribution. For instance, with a normal distribution.

$$p(\mu,\sigma^2|\{y_1,\ldots,y_n\}) = \frac{p(\{y_1,\ldots,y_n\}|\mu,\sigma^2)p(\mu,\sigma^2)}{\int\int p(\{y_1,\ldots,y_n\}|\mu,\sigma^2)p(\mu,\sigma^2)d\mu d\sigma^2}$$

The Metropolis Algorithm

The Metropolis algorithm is the most general implementation of Markov Chain Monte Carlo (MCMC).

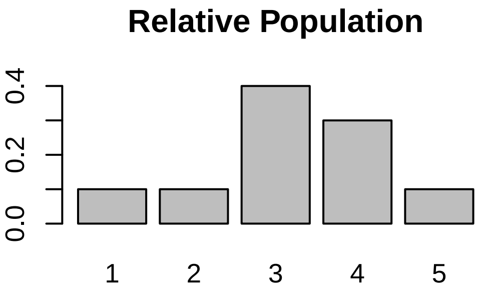
A Markov chain is a stochastic process where the the value of the next event only depends on the previous value.

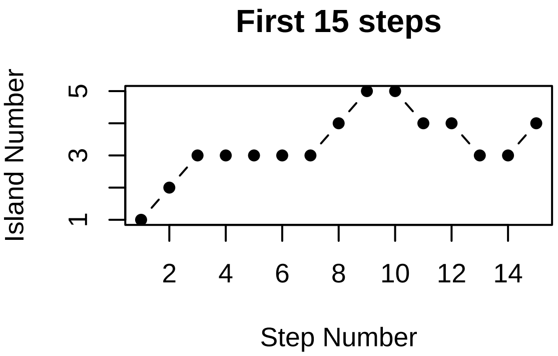
Politican stumbles across the Metropolis algorithm

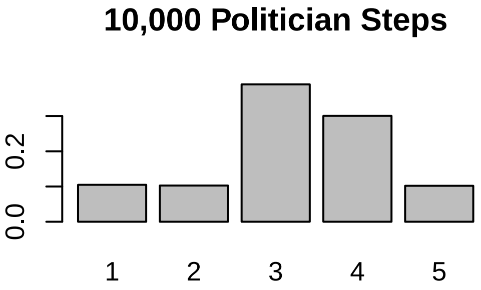
DBDA provides a nice intuitive overview of the Metropolis algorithm.

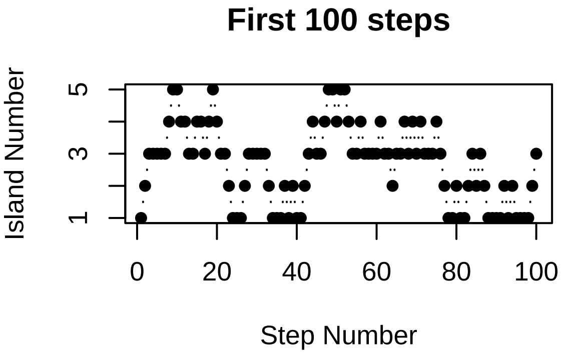
- The elected politician lives on a chain of islands and wants to stay in the public eye by traveling from island-to-island.
- At the end of day the politician needs to decide whether to:
  1. stay on the current island,
  2. move to the adjacent island to the west, or
  3. move to the adjacent island to the east.
- The politician's goal is to visit all islands proportional to their population, so most time is spent on the most populated islands. Unfortunately his office doesn't know the total population of the island chain. When visiting (or proposing to vist) an island, the politician can ask the local mayor for the population of the island.
- The politician has a simple heuristic for deciding whether to travel to a proposed island, by flipping a coin to determine whether to consider traveling east or west.

```r
par(mfcol=c(2,2))
# set up islands and relative population
num.islands <- 5
relative.population <- c(.1,.1,.4,.3,.1)
barplot(relative.population, names.arg = as.character(1:5), main='Relative Population')

# initialize politician
num.steps <- 10000
island.location <- rep(1,num.steps) # start at first island

# algorithm
for (i in 2:num.steps){
  direction <- sample(c('right','left'),1)
  if (direction == 'right'){
    proposed.island <- island.location[i-1] + 1
    if (proposed.island == 6) {
      island.location[i] <- island.location[i-1] #no island 6 exists, stay at island 5
    } else {
      prob.move <- relative.population[proposed.island] / relative.population[island.location[i-1]]
      if (runif(1) < prob.move){
        # move
        island.location[i] <- proposed.island
      } else{
        #stay
        island.location[i] <- island.location[i-1]
      }
    }
  }
  if (direction == 'left'){
    proposed.island <- island.location[i-1] - 1
    if (proposed.island == 0) {
      island.location[i] <- island.location[i-1] #no island 0 exists, stay at island 1
    } else {
      prob.move <- relative.population[proposed.island] / relative.population[island.location[i-1]]
      if (runif(1) < prob.move){
        # move
        island.location[i] <- proposed.island
      } else{
        #stay
        island.location[i] <- island.location[i-1]
      }
    }
  }
}
barplot(table(island.location) / num.steps, names.arg = as.character(1:5),
        main='10,000 Politician Steps')

plot(island.location[1:15], ylim=c(1,5),ylab='Island Number',xlab='Step Number',
     pch=16,type='b', main='First 15 steps')

plot(island.location[1:100], ylim=c(1,5),ylab='Island Number',xlab='Step Number',
     pch=16,type='b', main='First 100 steps')
```


Relative Population


First 15 steps


10,000 Politician Steps


First 100 steps

More details about the Markov chain

This procedure that we have described is a Markov chain. As such we can consider a few probabilities, let $l(i)$ be the politician's location at time i and assume the politician begins at island 5.:

- $Pr[l(1) = 5] = 1$
- $Pr[l(2) = 1] = 0$
- $Pr[l(2) = 2] = 0$
- $Pr[l(2) = 3] = 0$
- $Pr[l(2) = 4] = \frac{1}{2}$
- $Pr[l(2) = 5] = \frac{1}{2}$

We can also think about transition probabilities from state i to state j

More details about Metropolis

- The process to determine the next location to propose is known as the *proposal distribution*.

- Given a proposed site, we always move higher (larger value in the target function for the proposed cite) if we can.

- If a proposed cite is lower than the current position, the move is made with probability corresponding to the following ratio $\frac{p(\theta_{proposed})}{p(\theta_{current})}$, where $p(\theta)$ is the value of the target distribution at θ.

  The key elements of this process are:

1. Generate a random value from the proposal distribution to create $\theta_{proposed}$.

2. Evaluate the the target distribution to compute $\frac{\theta_{proposed}}{\theta_{current}}$.

3. Generate a random variable from uniform distribution to accept or reject proposal according to $p_{move} = \min\left(\frac{\theta_{proposed}}{\theta_{current}}, 1\right)$.

The ability to complete these three steps allows indirect sampling from the target distribution, even if it cannot be done directly (viz. `rnorm()`).

Generally our target distribution will be the posterior distribution, $p(\theta|\mathcal{D})$.

Furthermore, this process does not require a normalized distribution, which will mean we don't have to compute $\mathcal{D}$ in the denominator of Bayes rule as it will be the same for any θ and θ'. Hence evaluating the target distribution will amount to evaluating $p(\mathcal{D}|\theta) \times p(\theta)$.

The traveling politician example has:

- discrete positions (islands),
- one dimension (east-west),
- and a proposal distribution of one step east or west.

  This procedure also works more generally for:

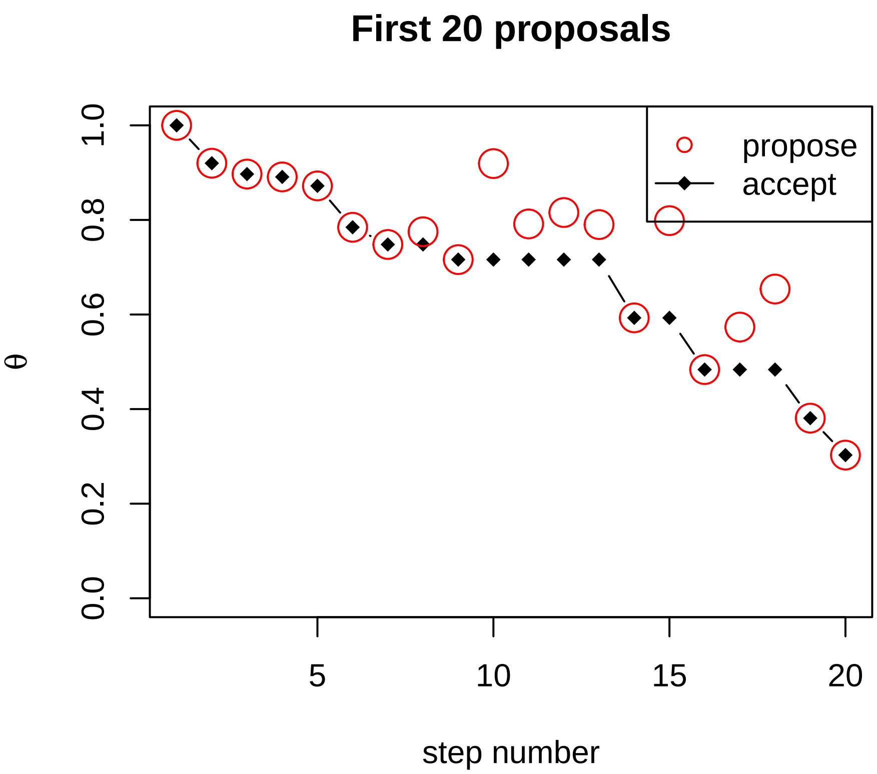
- continuous values (consider the probability parameter in occupancy model),
- any number of dimensions,
- and more general proposal distributions.

Metropolis Sampler for Beta Prior and Bernoulli likelihood

We will soon see learn about JAGS for fitting Bayesian models, but these algorithms can also be written directly in R code.

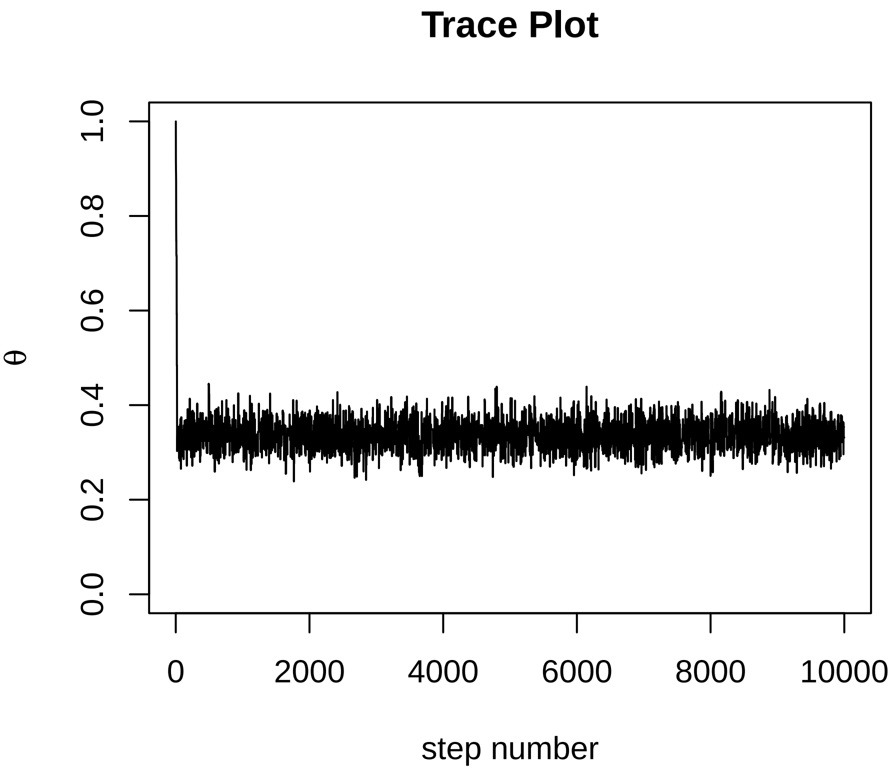
This will be demonstrated on the willow tit dataset and the MCMC results will be compared with the analytical solution. In most cases, analytical solutions for the posterior are not possible and MCMC is typically used to make inferences from the posterior.

```r
# set prior parameters for beta distribution
a.prior <- 1
b.prior <- 1

# read in data
birds <- read.csv('http://www.math.montana.edu/ahoegh/teaching/stat491/data/willowtit2013.csv')
y <- birds$birds
N <- nrow(birds) # count number of trials
z <- sum(birds$birds)

# initialize algorithm
num.sims <- 10000
sigma.propose <- .1 # standard deviation of normal random walk proposal distribution
theta.accept <- rep(0, num.sims)
theta.current <- rep(1, num.sims)
theta.propose <- rep(1, num.sims)

for (i in 2:num.sims){
  # Step 1, propose new theta
  while(theta.propose[i] <= 0 | theta.propose[i] >= 1){
      theta.propose[i] <- theta.current[i-1] + rnorm(n = 1, mean = 0, sd = sigma.propose)
  }

  # Step 2, compute p.move - note this is on a log scale
  log.p.theta.propose <- sum(dbinom(y, 1, theta.propose[i], log = T)) +
    dbeta(theta.propose[i], a.prior, b.prior, log = T)
  log.p.theta.current <- sum(dbinom(y, 1, theta.current[i-1], log = T)) +
    dbeta(theta.current[i-1], a.prior, b.prior, log = T)
  log.p.move <- log.p.theta.propose - log.p.theta.current

  # Step 3, accept with probability proportional to p.move - still on log scale
  if (log(runif(1)) < log.p.move){
    theta.current[i] <- theta.propose[i]
    theta.accept[i] <- 1
  } else{
    theta.current[i] <- theta.current[i-1]
  }
}
par(mfcol=c(1,1))
plot(theta.current[1:20], type = 'b', pch=18, ylim=c(0,1), ylab = expression(theta),
     main = 'First 20 proposals', xlab='step number')
points(theta.propose[1:20], pch=1, col='red', cex=2)
legend('topright', legend = c('propose','accept'),col=c('red','black'), lty =c(NA,1), pch = c(1,18))
```



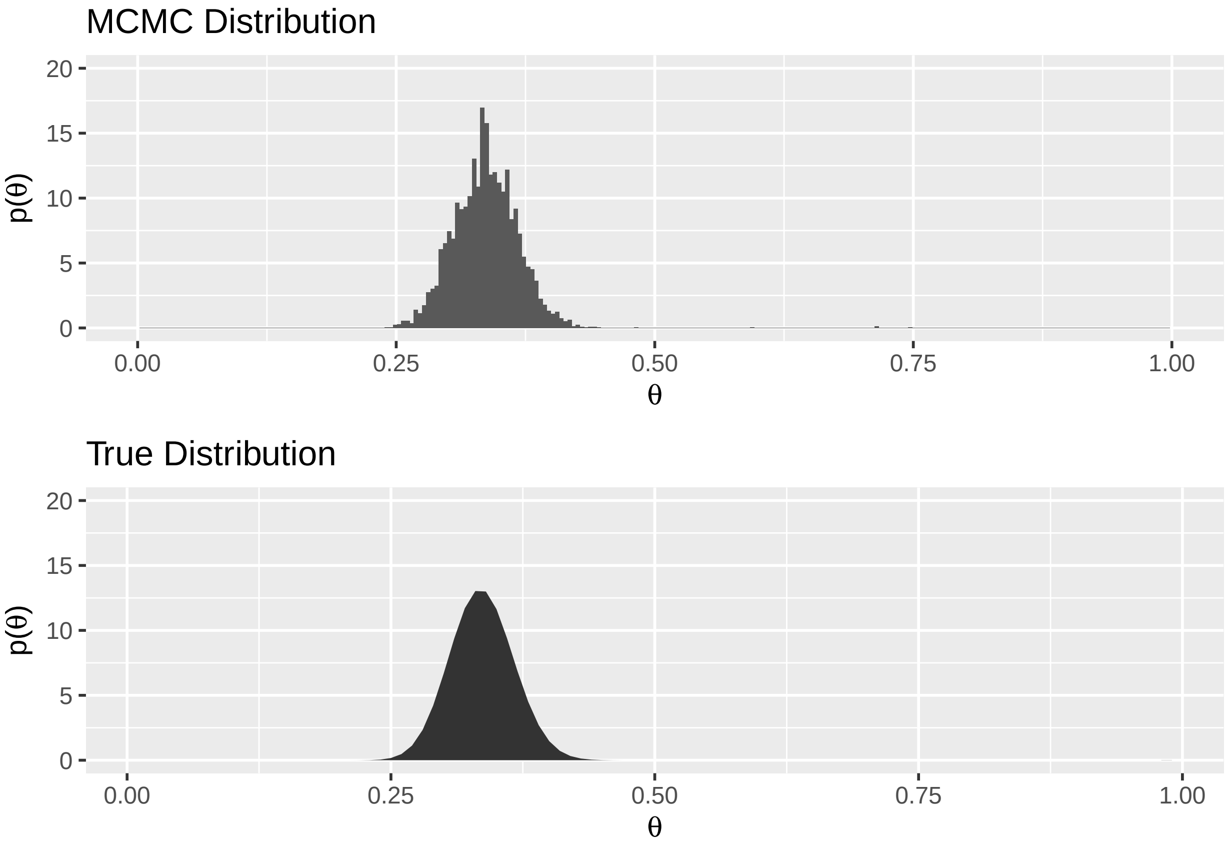
Now after viewing the first twenty steps, consider all steps.

```
plot(theta.current, type = 'l', ylim=c(0,1), ylab = expression(theta),
     main = 'Trace Plot', xlab='step number')
```



Now look at a histogram depiction of the distribution.

```r
par(mfrow=c(1,1))
library(ggplot2)
library(gridExtra)
df <- data.frame(theta.current)
hist.mcmc <- ggplot(df) + geom_histogram(aes(x=theta.current,y=..density..), bins = 250) +
  xlab(expression(theta)) + ylab(expression(paste('p(',theta,')',sep=''))) +
  ggtitle('MCMC Distribution') + xlim(0,1) + ylim(0,20)

theta <- seq(0.01,0.99, by = .01)
p.theta <- dbeta(theta, a.prior + z, b.prior + N -z)
true.df <- data.frame(theta, p.theta)
curve.true <- ggplot(true.df) + geom_polygon(aes(x=theta, y=p.theta)) + xlab(expression(theta)) +
  ylab(expression(paste('p(',theta,')',sep=''))) + ggtitle('True Distribution') + ylim(0,20)
grid.arrange(hist.mcmc, curve.true, nrow=2)
```



In this case, we see that the distributions look very similar. In general with MCMC there are three goals:

1. The values in the chain must be **representative** of the posterior distribution.

2. The chain should be of sufficient size so estimates are **accurate** and **stable**.

3. The chain should be generated **efficiently**.

JAGS

JAGS is a software package for conducting MCMC. We will run this through R, but note you also need to download JAGS to your computer. You will not be able to reproduce this code or run other JAGS examples if JAGS has not been installed.

There are a few common examples for running JAGS code, which will be illustrated below:

1. Load the data and place it in a list object. The list will eventually be passed to JAGS.

```
library(rjags)
```

```
## Loading required package: coda

## Linked to JAGS 4.3.0

## Loaded modules: basemod,bugs
```

```
library(runjags)
birds <- read.csv('http://www.math.montana.edu/ahoegh/teaching/stat491/data/willowtit2013.csv')
y <- birds$birds
N <- nrow(birds) # count number of trials
z <- sum(birds$birds)
dataList = list(y = y, Ntotal = N)
```

2. Specify the model as a text variable. While the code looks vaguely familiar, it to is executed in JAGS. The model statement contains the likelihood piece, $p(y|\theta)$, written as a loop through the N Bernoulli observations and the prior, $p(\theta)$. Finally the model is bundled as a .txt object.

```
modelString = "
  model {
    for ( i in 1:Ntotal ) {
      y[i] ~ dbern( theta ) # likelihood
    }
    theta ~ dbeta( 1 , 1 ) # prior
  }
"
writeLines( modelString, con='TEMPmodel.txt')
```

3. Initialize the chains by specifying a starting point. This is akin to stating which island the politician will start on. It is often advantageous to run a few chains with different starting points to verify that they have the same end results.

```
initsList <- function(){
  # function for initializing starting place of theta
  # RETURNS: list with random start point for theta
  return(list(theta = runif(1)))
}
```

4. Generate MCMC chains. Now we call the JAGS code to run the MCMC. The `jags.model()` function takes:
    - a file containing the model specification
    - the data list
    - the list containing the initialized starting points
    - the function also permits running multiple chains, `n.chain`,
    - `n.adapt` works to tune the algorithm.

```
jagsModel <- jags.model( file = "TEMPmodel.txt", data = dataList, inits =initsList,
                         n.chains =3, n.adapt = 500)
```

```
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 242
##    Unobserved stochastic nodes: 1
##    Total graph size: 245
##
## Initializing model
```

```
update(jagsModel, n.iter = 500)
```

The `update` statement results in what is called the burn in period, which is essentially tuning the algorithm and those samples are ultimately discarded. Now we can run the algorithm for a little longer (let the politician walk around).

```
codaSamples <- coda.samples( jagsModel, variable.names = c('theta'), n.iter =3334)
```

5. Examine the results. Finally we can look at our chains to evaluate the results.

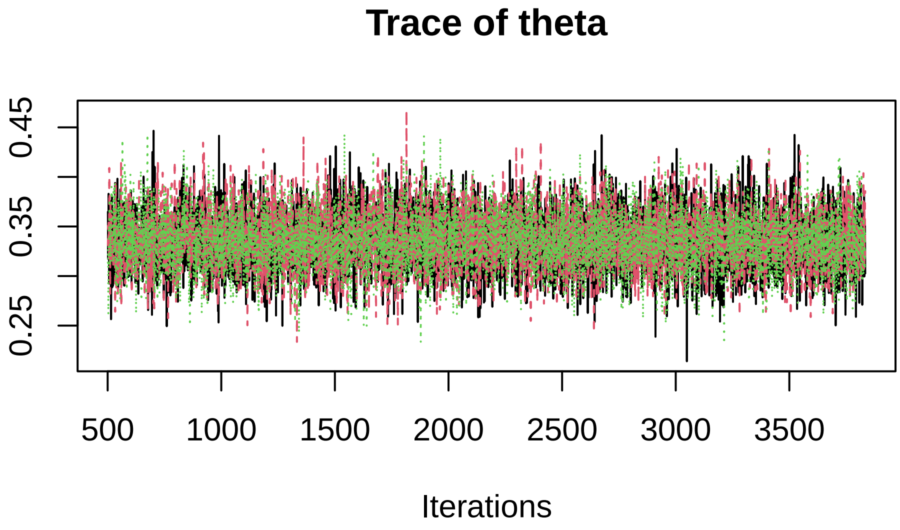
```
HPDinterval(codaSamples)
```

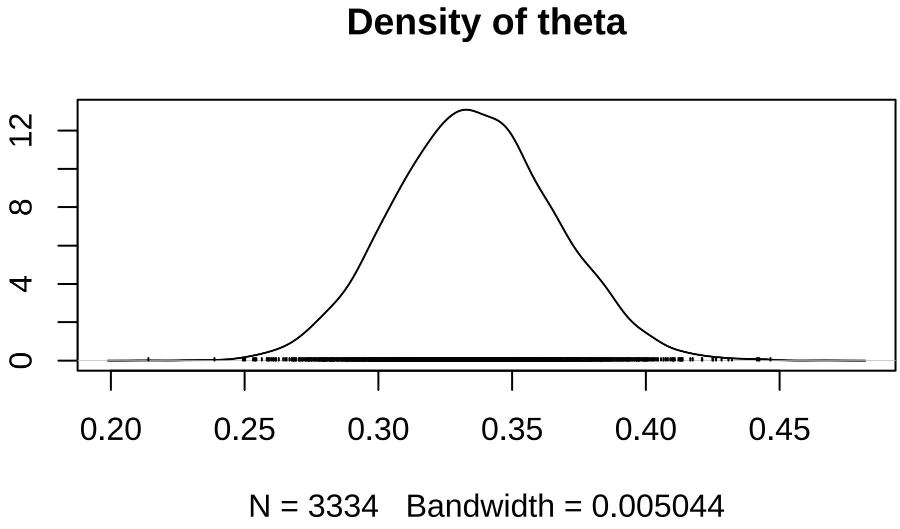
```
## [[1]]
##           lower     upper
## theta 0.2767714 0.3941148
## attr(,"Probability")
## [1] 0.94991
##
## [[2]]
##           lower     upper
## theta 0.2780405 0.3941263
## attr(,"Probability")
## [1] 0.94991
##
## [[3]]
##           lower     upper
## theta 0.2776954 0.3922069
## attr(,"Probability")
## [1] 0.94991
```

```
summary(codaSamples)
```

```
##
## Iterations = 501:3834
## Thinning interval = 1
## Number of chains = 3
## Sample size per chain = 3334
##
## 1. Empirical mean and standard deviation for each variable,
##    plus standard error of the mean:
##
##           Mean             SD       Naive SE Time-series SE
##      0.3358308      0.0300242      0.0003002      0.0003002
##
## 2. Quantiles for each variable:
##
##   2.5%    25%    50%    75%  97.5%
## 0.2784 0.3151 0.3353 0.3556 0.3953
```

```
par(mfcol=c(2,1))
traceplot(codaSamples)
densplot(codaSamples)
```


Trace of theta
Iterations


Density of theta
N = 3334   Bandwidth = 0.005044