MCMC DEMO

MCMC DEMO

1. Exact Posterior

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 using a dataset that determines if birds (the willow tit) are observed in a spatial grid square. 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.

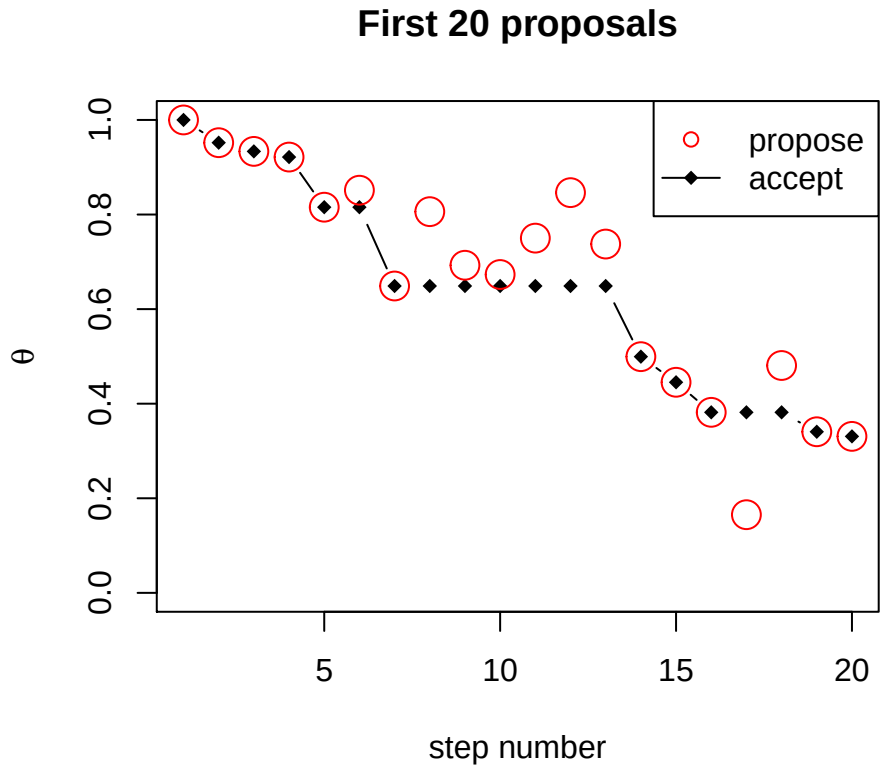
```
# set prior parameters for beta distribution
a.prior <- 1
b.prior <- 1

# 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)
```

The true posterior distribution is beta(82, 162).

2. Metropolis Sampler for Beta Prior and Bernoulli likelihood

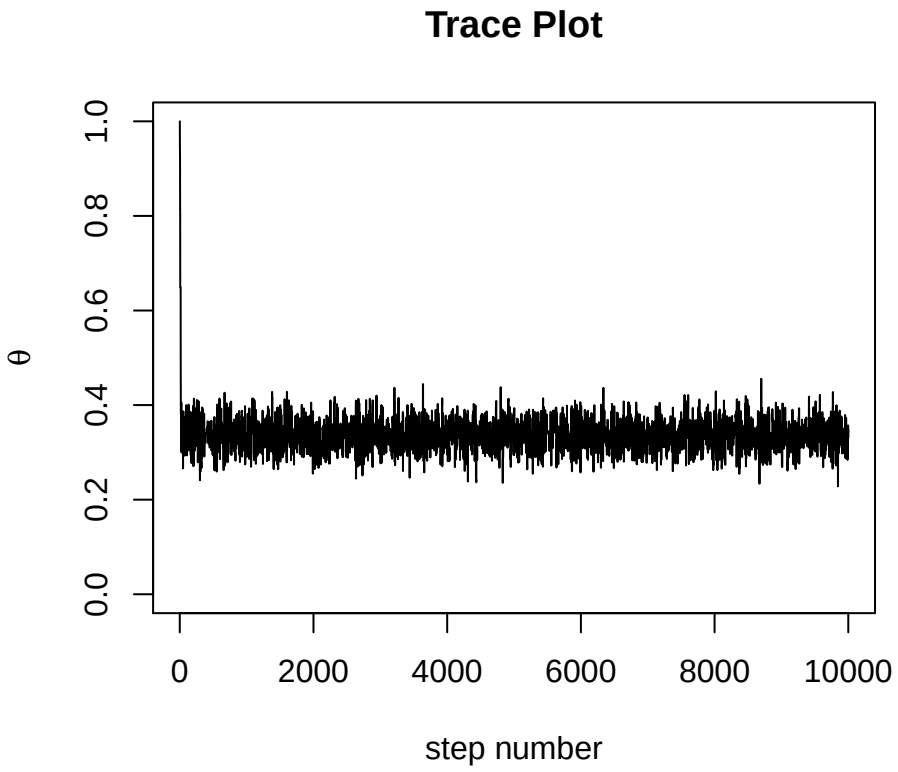
```
# set prior parameters for beta distribution
a.prior <- 1
b.prior <- 1

# 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.

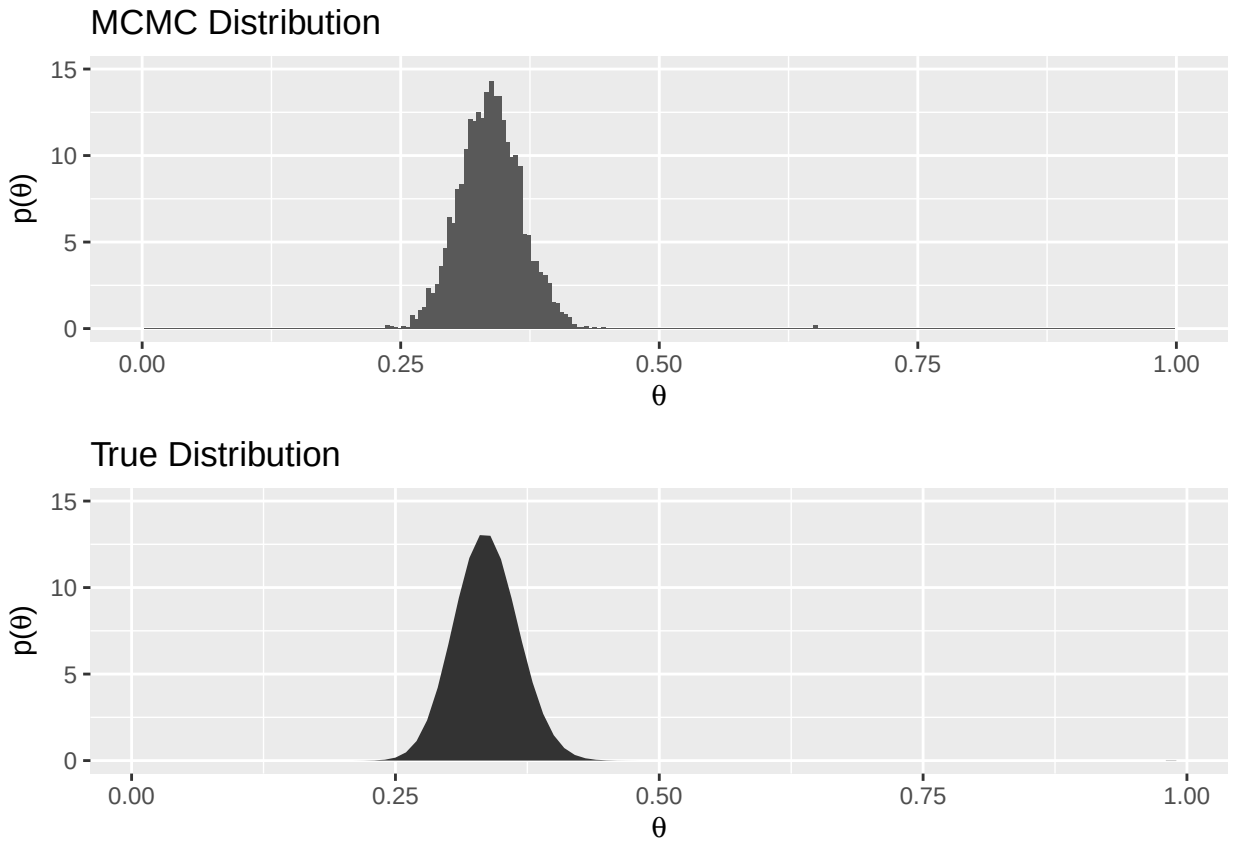
```
plot(theta.current, type = 'l', ylim=c(0,1), ylab = expression(theta),
     main = 'Trace Plot', xlab='step number')
```


Trace Plot
θ
0.0
0.2
0.4
0.6
0.8
1.0
0
2000
4000
6000
8000
10000
step number

Now look at a histogram depiction of the distribution.

```
par(mfrow=c(1,1))
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(0,15)

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(0,15)
grid.arrange(hist.mcmc, curve.true, nrow=2)
```

```
## Warning: Removed 2 rows containing missing values (geom_bar).
```



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**.

3. MCMC with 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)
```

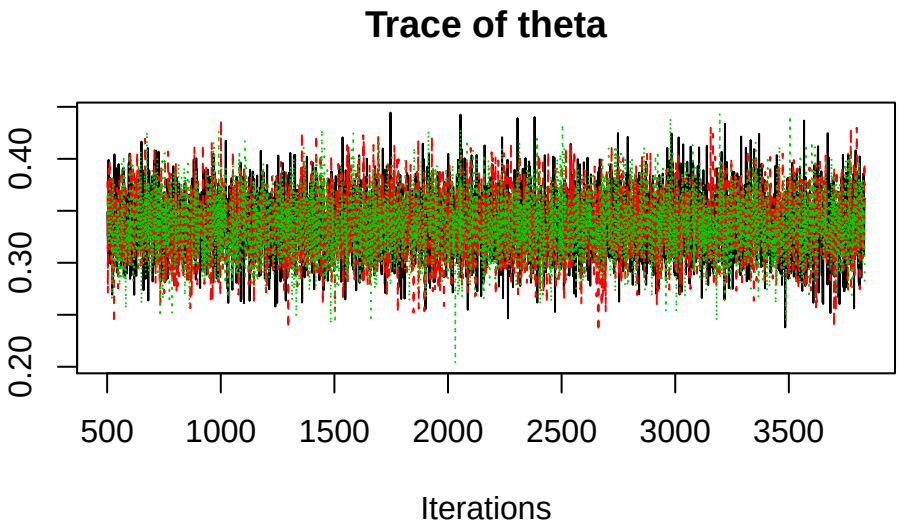
```
## [[1]]
##           lower     upper
## theta 0.2806796 0.3972981
## attr(,"Probability")
## [1] 0.94991
##
## [[2]]
##           lower     upper
## theta 0.2786207 0.3970706
## attr(,"Probability")
## [1] 0.94991
##
## [[3]]
##           lower     upper
## theta 0.2776068 0.3939864
## attr(,"Probability")
## [1] 0.94991
```

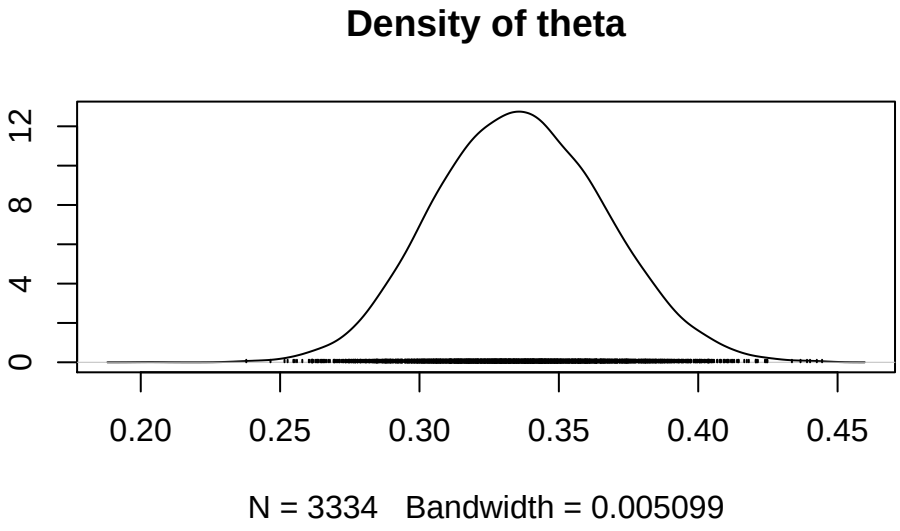
```
summary(codaSamples)
```

```
##
## 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
##      0.3362798      0.0303530      0.0003035      0.0003067
##
## 2. Quantiles for each variable:
##
##   2.5%    25%    50%    75%  97.5%
```

```
## 0.2790 0.3151 0.3357 0.3570 0.3967
```

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





4. MCMC with stan

Stan is an alternative to JAGS for fitting MCMC. Stan implements a slightly different approach for proposing new locations, known as Hamiltonian Monte Carlo.

We may look at Stan later in the class, but the programming is more involved than JAGS. Stan uses C++ as the base code. Additionally, the data and parameters are defined separately. Stan also permits vectorized operations, such as `y~bernoulli(theta)`.

```
library(rstan)
```

```
## Loading required package: StanHeaders
```

```
## rstan (Version 2.18.2, GitRev: 2e1f913d3ca3)
```

```
## For execution on a local, multicore CPU with excess RAM we recommend calling
## options(mc.cores = parallel::detectCores()).
## To avoid recompilation of unchanged Stan programs, we recommend calling
## rstan_options(auto_write = TRUE)
```

```
##
## Attaching package: 'rstan'
```

```
## The following object is masked from 'package:runjags':
##
##     extract
```

```
## The following object is masked from 'package:coda':
##
##     traceplot
```

```
# specify model

modelString = "
  data {
    int<lower=0> N;
    int y[N] ; // y is a length-N vector of integers
  }
  parameters {
    real<lower=0,upper=1> theta ;
  }
  model {
    theta ~ beta(1,1) ;
    y ~ bernoulli(theta) ;
  }
"

stanDSO <- stan_model(model_code = modelString)

# reuse bird dataset
birds <- read.csv('http://www.math.montana.edu/ahoegh/teaching/stat491/data/willowtit2013.csv')
y <- birds$birds
N <- nrow(birds) # count number of trials
dataList <- list(y=y, N=N)

# run code in stan
stanFit <- sampling(object=stanDSO, data=dataList, chains=3,
                    iter=10000, warmup =200, thin=1)
```

```
##
## SAMPLING FOR MODEL 'c681021e1b4e5238757f28e97116ddc0' NOW (CHAIN 1).
## Chain 1:
## Chain 1: Gradient evaluation took 1.3e-05 seconds
## Chain 1: 1000 transitions using 10 leapfrog steps per transition would take 0.13 seconds.
## Chain 1: Adjust your expectations accordingly!
## Chain 1:
## Chain 1:
## Chain 1: Iteration:    1 / 10000 [  0%]  (Warmup)
## Chain 1: Iteration:  201 / 10000 [  2%]  (Sampling)
## Chain 1: Iteration: 1200 / 10000 [ 12%]  (Sampling)
## Chain 1: Iteration: 2200 / 10000 [ 22%]  (Sampling)
## Chain 1: Iteration: 3200 / 10000 [ 32%]  (Sampling)
## Chain 1: Iteration: 4200 / 10000 [ 42%]  (Sampling)
## Chain 1: Iteration: 5200 / 10000 [ 52%]  (Sampling)
## Chain 1: Iteration: 6200 / 10000 [ 62%]  (Sampling)
## Chain 1: Iteration: 7200 / 10000 [ 72%]  (Sampling)
## Chain 1: Iteration: 8200 / 10000 [ 82%]  (Sampling)
## Chain 1: Iteration: 9200 / 10000 [ 92%]  (Sampling)
## Chain 1: Iteration: 10000 / 10000 [100%]  (Sampling)
## Chain 1:
## Chain 1:  Elapsed Time: 0.003259 seconds (Warm-up)
## Chain 1:                0.131822 seconds (Sampling)
## Chain 1:                0.135081 seconds (Total)
## Chain 1:
##
## SAMPLING FOR MODEL 'c681021e1b4e5238757f28e97116ddc0' NOW (CHAIN 2).
## Chain 2:
## Chain 2: Gradient evaluation took 5e-06 seconds
## Chain 2: 1000 transitions using 10 leapfrog steps per transition would take 0.05 seconds.
## Chain 2: Adjust your expectations accordingly!
## Chain 2:
## Chain 2:
## Chain 2: Iteration:    1 / 10000 [  0%]  (Warmup)
## Chain 2: Iteration:  201 / 10000 [  2%]  (Sampling)
## Chain 2: Iteration: 1200 / 10000 [ 12%]  (Sampling)
## Chain 2: Iteration: 2200 / 10000 [ 22%]  (Sampling)
## Chain 2: Iteration: 3200 / 10000 [ 32%]  (Sampling)
## Chain 2: Iteration: 4200 / 10000 [ 42%]  (Sampling)
## Chain 2: Iteration: 5200 / 10000 [ 52%]  (Sampling)
## Chain 2: Iteration: 6200 / 10000 [ 62%]  (Sampling)
## Chain 2: Iteration: 7200 / 10000 [ 72%]  (Sampling)
## Chain 2: Iteration: 8200 / 10000 [ 82%]  (Sampling)
## Chain 2: Iteration: 9200 / 10000 [ 92%]  (Sampling)
## Chain 2: Iteration: 10000 / 10000 [100%]  (Sampling)
## Chain 2:
## Chain 2:  Elapsed Time: 0.003523 seconds (Warm-up)
## Chain 2:                0.137972 seconds (Sampling)
## Chain 2:                0.141495 seconds (Total)
## Chain 2:
##
## SAMPLING FOR MODEL 'c681021e1b4e5238757f28e97116ddc0' NOW (CHAIN 3).
## Chain 3:
## Chain 3: Gradient evaluation took 5e-06 seconds
```

```
## Chain 3: 1000 transitions using 10 leapfrog steps per transition would take 0.05 seconds.
## Chain 3: Adjust your expectations accordingly!
## Chain 3:
## Chain 3:
## Chain 3: Iteration:    1 / 10000 [  0%]  (Warmup)
## Chain 3: Iteration:  201 / 10000 [  2%]  (Sampling)
## Chain 3: Iteration: 1200 / 10000 [ 12%]  (Sampling)
## Chain 3: Iteration: 2200 / 10000 [ 22%]  (Sampling)
## Chain 3: Iteration: 3200 / 10000 [ 32%]  (Sampling)
## Chain 3: Iteration: 4200 / 10000 [ 42%]  (Sampling)
## Chain 3: Iteration: 5200 / 10000 [ 52%]  (Sampling)
## Chain 3: Iteration: 6200 / 10000 [ 62%]  (Sampling)
## Chain 3: Iteration: 7200 / 10000 [ 72%]  (Sampling)
## Chain 3: Iteration: 8200 / 10000 [ 82%]  (Sampling)
## Chain 3: Iteration: 9200 / 10000 [ 92%]  (Sampling)
## Chain 3: Iteration: 10000 / 10000 [100%]  (Sampling)
## Chain 3:
## Chain 3:  Elapsed Time: 0.003508 seconds (Warm-up)
## Chain 3:                0.134982 seconds (Sampling)
## Chain 3:                0.13849 seconds (Total)
## Chain 3:
```

```
#convert to coda object
stanFit
```

```
## Inference for Stan model: c681021e1b4e5238757f28e97116ddc0.
## 3 chains, each with iter=10000; warmup=200; thin=1;
## post-warmup draws per chain=9800, total post-warmup draws=29400.
##
##          mean se_mean   sd    2.5%     25%     50%     75%   97.5% n_eff
## theta    0.34    0.00 0.03    0.28    0.32    0.34    0.36    0.40 10166
## lp__  -156.27    0.01 0.71 -158.29 -156.44 -156.00 -155.82 -155.77 13826
##       Rhat
## theta    1
## lp__     1
##
## Samples were drawn using NUTS(diag_e) at Mon Sep  9 10:20:21 2019.
## For each parameter, n_eff is a crude measure of effective sample size,
## and Rhat is the potential scale reduction factor on split chains (at
## convergence, Rhat=1).
```

```
summary(stanFit)
```

```
## $summary
##               mean      se_mean         sd         2.5%         25%
## theta    0.3363106 0.0002997773 0.03022603    0.2788377    0.315586
## lp__  -156.2698265 0.0060337327 0.70947921 -158.2944029 -156.444444
##                50%         75%        97.5%    n_eff     Rhat
## theta    0.3358548    0.356745    0.3963229 10166.35 1.000272
## lp__  -155.9994260 -155.817183 -155.7679622 13826.34 1.000289
##
## $c_summary
## , , chains = chain:1
##
##          stats
## parameter         mean         sd         2.5%          25%          50%
```

```
##     theta    0.3360821 0.03064102    0.2778046    0.3151149    0.3355508
##     lp__  -156.2837397 0.73185543 -158.3730071 -156.4543397 -156.0085004
##          stats
## parameter          75%        97.5%
##     theta    0.3571191    0.3974131
##     lp__  -155.8193543 -155.7680118
##
## , , chains = chain:2
##
##          stats
## parameter         mean         sd        2.5%          25%          50%
##     theta    0.3364831 0.02960949    0.280128    0.3161667    0.3359017
##     lp__  -156.2491549 0.68764574 -158.203774 -156.4096451 -155.9897016
##          stats
## parameter          75%        97.5%
##     theta    0.3564448    0.3951374
##     lp__  -155.8133907 -155.7678768
##
## , , chains = chain:3
##
##          stats
## parameter         mean         sd         2.5%          25%          50%
##     theta    0.3363666 0.03041951    0.2786245    0.3153862    0.3360839
##     lp__  -156.2765850 0.70784835 -158.2702931 -156.4672817 -156.0029382
##          stats
## parameter          75%        97.5%
##     theta    0.3568643    0.3964231
##     lp__  -155.8179655 -155.7679739
```

5. Wrap Up

1. How do the results of the methods compare?

2. Of the MCMC approaches, which do you find the most intuitive?

3. Of the MCMC approaches, which do you find the easiest to implement?