Introduction to JAGS

This document draws heavily from https://nthobbs50.github.io/BayesNSF/content/lectures/JAGSPrimerMCMCVis.pdf

In the following, you will most just read the code and accompanying text. Then, whenever you see a section header with **this color**, you should copy/paste or write/run code into your own `.Rmd` or `.qmd` file as practice.

Solutions to checks and practice are found at the end of the document!

Installing JAGS (Just Another Gibbs Sampler)

MAC OS

Go to https://sourceforge.net/projects/mcmc-jags/files/ and download `JAGS-4.3.0.dmg` to get the disk mounting image. Click on the GREEN button to download the package, and go through the installation as you would any other Mac software.

Once the installation process has completed, install the package `rjags` either in the Console using `install.packages("rjags")` or in the Packages pane. Then load the package as usual:

```
library(rjags)
```

Windows

Go to https://sourceforge.net/projects/mcmc-jags/files/ and download `JAGS-4.3.0.exe`. Occasionally, Windows users have problems loading `rjags` from R after everything has been installed properly. This problem usually occurs because they have more than one version of R resident on their computers (wisely, Mac OS will not allow that). So, if you can't seem to get `rjags` to run after a proper install, then uninstall all versions of R, re-install the latest version, install the latest version of `rjags` and the version of JAGS that matches it.

Then load the package as you would any other package:

```
library(rjags)
```

Motivation

Suppose we would like to fit a Bayesian model to our tennis serve data (treating all players as coming from the same population), where both the mean and standard deviation are unknown, and we place independent priors on both parameters. We have learned that for specific choices of priors for the mean θ and variance σ^2, we can derive full conditionals and run a Gibbs sampler to approximate the joint posterior of (θ, σ^2). However, what happens if we'd like use a different set of priors? We could instead turn to Metropolis instead of the Gibbs sampler. But as we've seen, sometimes the parameter space isn't convenient to work with for a symmetric proposal distribution, or "tuning" the algorithm takes a lot of work. Is there a way to get around some of these issues?

We will leverage `JAGS` which will automatically select an appropriate MCMC sampling algorithm for a particular Bayesian model. This is so nice! We will demonstrate with the following model.

Suppose we have the following sampling model:

$$Y_i|\theta, \sigma^2 \overset{iid}{\sim} N(\theta, \sigma^2), \qquad i = 1, \dots, n$$

I will use independent priors for θ and σ^2:

$$\theta \sim N(\mu_0, \sigma_0^2) \qquad \sigma^2 \sim \text{Gamma}(a, b)$$

(Note, this isn't allowing for semi-conjugacy as it's the variance σ^2 that's getting the Gamma prior, not the precision $1/\sigma^2$.)

How do we tell JAGS that we have the Normal sampling model with this choice of priors? We will have to write a script in syntax particular to JAGS, though it looks quite similar to that of `R`.

Check 1: based on this model statement, is this a complete pooling, no pooling, or hierarchical model?

Defining a JAGS model

Describe the model by a script

To begin, we write a script that corresponds to our model. *The curly braces and quotes all matter here!* The particular statistical model goes in the `model{ <this stuff> }` section of the following code.

As in usual `R`, the pound signs are comments, so we can clearly see where in the script I am defining the sampling model and where I am defining my priors.

```
{
sink("normal_mod.R")
cat("
model {
## sampling model
for (i in 1:N) {
   y[i] ~ dnorm(theta, tau2)
}

## priors
theta ~ dnorm(mu0, phi0)
sigma2 ~ dgamma(a, b)
tau2 <- 1/sigma2
}
", fill = T)
sink()
}
```

The `sink()` function means that this code is saved as an `R` file called `normal_mod.R` in the current working directory. **When you write your own JAGS model, you should only change 1) the name of the file (i.e. `sink(<file name>)`), and 2) the code within `model{ }`.** Every time you run this code, you will save over any file with the same file name (in this case, `normal_mod.R`).

Note that this script closely resembles the statement of the model. In the sampling model part of the script, the loop structure starting with `for (i in 1:N)` is used to assign the distribution of each value in the data vector `y` the same Normal distribution (i.e. our sampling model), represented by `dnorm`. The tilde `~` is read as "is distributed as". Notice that we have to index `y` using the `[i]` to tell JAGS the specific distribution for y_i. JAGS isn't vectorized.

Then, we come to the part of the script where I specify priors. Here, we define the priors for μ and σ^2 using `dnorm` and `dgamma`, respectively. Note the code `tau2 <- 1/sigma2`. The

arrow terminology tells JAGS to create `tau2` as $\tau^2 = \frac{1}{\sigma^2}$. Why do I do this? This is because JAGS parameterizes the Normal distribution in terms of **precision**, not standard deviation nor variance. That's why in the sampling model portion, I write `dnorm(theta, tau2)` instead of `dnoram(theta, sigma2)`. You should also notice that the prior for θ has a *precision* of `phi0`.

Define the data and prior parameters

Great! Now we're ready to actually run this model on our data. We have to define pass in relevant quantities from the data and provide values for the parameters of the priors. In particular, JAGS is expecting inputs for the following: `y`, `N`, `mu0`, `phi0`, `a`, `b`. The list below, named `data_ls`, is used to collect all of these inputs (using the tennis serves data from a previous homework).

```
y <- tennis_dat$Speed_MPH
n_y <- length(y)

data_ls <- list("y" = y, "N" = n_y,
                "mu0" = 100, "phi0" = 0.04,
                "a" = 25, "b" = 1)
```

Note the use of quotes. It's important to recognize that the terms in quotes on the left hand side of the equation correspond to the values in the JAGS script, and the terms on the right-hand side correspond to values in `R`. For example, our JAGS script has `N` as the number of observations, but I've defined that in `R` as `n_y`.

Check 2: based on this code, what exactly are the priors I am choosing for this model?

Note: I could have totally specified the prior parameter values in the script itself. However, specifying `a` and `b` and the other prior values in `data_ls` to be passed into the script makes my code more reproducible!

Running JAGS

Initialize

All MCMC algorithms needs to be initialized. You can specify intitial values if you'd like. If no initial values are specified, then JAGS will select initial values based on the prior. For now, we won't worry about setting initial values and will let JAGS do it for us. However, in certain problems/models, where you start the sampler is extremely important! This is especially true in models where latent variables are involved.

Let JAGS determine MCMC algorithm

Now that the model and data have been defined, we are ready for MCMC.

```
n_chain <- 2
jm <- jags.model("normal_mod.R",
                 data = data_ls,
                 n.chains = n_chain)
```

The `jm <- jags.model...` statement sets up the MCMC chain. Its first argument is the name of the file containing the JAGS code. *Note that my file resides in the current working directory in this example. Otherwise, you would need to specify the full path name.* The next two expressions specify where the data come from (i.e. `data_ls` which we created) and how many chains `n.chains` to create (i.e., how many independent samplers we should run). We have thus far been running a single chain for convenience. Specifying more than one chain is typically good, because each chain gets initialized at a different value so we have more opportunities to explore the parameter space (and ensure that we reach convergence).

During this stage, JAGS figures out which MCMC sampling algorithm it will use to (approximate) the posterior. It might do exact posterior sampling, Gibbs, Metropolis, or something else depending on the combination of prior and data model.

Burn-in

After JAGS has determined the appropriate sampler, we actually run the code for some amount of burn-in. This code below specifies the burn-in amount via `n.iter`, i.e. how many samples to throw away before beginning to save values in each chain. This code "updates" the `jm` object by executing the burn-in.

```
B <- 5000
update(jm, n.iter = B)
```

Sampling from the posterior

After burn-in, we are finally ready to draw/store samples from the posterior distribution.

```
jags_out <- coda.samples(jm,
                         variable.names = c("theta", "sigma2"),
                         n.iter = B)
```

This code actually runs the MCMC using the `coda.samples()` function, and stores the output as an MCMC list (more about that soon) called `jags_out`. The first argument is the name of the model from the previous step, `jm`. The second argument, `variable.names`, tells JAGS which values you'd like to "monitor", i.e. which variables you would like the posteriors of. This is to save memory. If you don't specify any variables, you won't be able to do much! Note that that the names of the variables must be exactly as written in the your script defining the model. In this case, I want the posteriors of μ and σ^2. I could have also asked for the precision τ^2.

Finally, `n.iter` specifies how many iterations we'd like to run in *each* chain after burn-in. Although the argument name `n.iter` is used in both `update()` and `coda.samples()`, you can specify different numbers of iterations for both.

Check 3: after running the code, how many samples should we have of each parameter?

Run it yourself

So now, in total, we have the following code. Go ahead and copy and paste this into an `.Rmd` or `.qmd` file, and run it all by knitting the document or simply running locally (note you'll have to load in the tennis data beforehand). If you just run locally, you'll notice that as your run the `jags.model()` and `update()` and `coda.samples()` functions, you'll get little progress bars in your console. That's good! That means things are running!

Comment the code as you'd like!

After running the code, you should see a file called `normal_mod.R` in the folder where your working directory is pointing to.

```
library(rjags)
{
sink("normal_mod.R")
cat("
model {
## sampling model
for (i in 1:N) {
   y[i] ~ dnorm(theta, tau2)
}

## priors
theta ~ dnorm(mu0, phi0)
sigma2 ~ dgamma(a, b)
tau2 <- 1/sigma2
}
", fill = T)
sink()
}
tennis_dat <- readRDS("tennis_serves.Rda")
y <- tennis_dat$Speed_MPH
n_y <- length(y)
data_ls <- list("y" = y, "N" = n_y,
                "mu0" = 100, "phi0" = 0.04,
                "a" = 25, "b" = 1)
n_chain <- 2
jm <- jags.model("normal_mod.R", data = data_ls, n.chains = n_chain)
```

```
Compiling model graph
   Resolving undeclared variables
   Allocating nodes
Graph information:
   Observed stochastic nodes: 300
   Unobserved stochastic nodes: 2
   Total graph size: 309

Initializing model
```

```
B <- 5000
update(jm, n.iter = B)
jags_out <- coda.samples(jm,
                         variable.names = c("theta", "sigma2"),
                         n.iter = B)
```

Output from JAGS

The `coda.samples()` function in the last line produces output in the form of an `mcmc.list` object, sometimes referred to as a coda object. What does this mean? Let's take a look at the first few values:

```
head(jags_out)
```

```
[[1]]
Markov Chain Monte Carlo (MCMC) output:
Start = 6001
End = 6007
Thinning interval = 1
        sigma2    theta
[1,] 103.66966 106.3623
[2,] 105.38932 105.8220
[3,] 102.31322 106.7747
[4,]  98.32038 107.1256
[5,] 104.90416 105.7967
[6,] 101.42569 106.4491
[7,]  97.35865 105.9876

[[2]]
Markov Chain Monte Carlo (MCMC) output:
Start = 6001
End = 6007
Thinning interval = 1
        sigma2    theta
[1,]  95.45099 107.0273
[2,] 115.92905 106.6055
[3,] 115.84605 105.9112
[4,]  91.40939 107.0472
[5,]  98.28039 105.3330
[6,] 105.69049 105.8803
[7,]  99.08535 106.2880

attr(,"class")
[1] "mcmc.list"
```

The output of coda is a list of matrices, where each matrix contains the output of a single chain for all parameters being estimated. Parameter values are stored in the columns of the matrix;

values for one iteration of the chain are stored in each row. Since we specified we wanted 2 chains, the output produces a list of two matrices.

Practice

Load in `tidyverse` first!

1. Convert your coda object `jags_out` into a data frame using the following code:

   `posts_df <- data.frame(do.call(rbind, jags_out)) |> add_column(chain = factor(rep(1:n_chain, each = B)))`.

   This code means "row bind all the matrices in `jags_out` (in this case just one) into one large matrix, then convert into a data frame". So the samples from the second chain get appended to the end of the samples from the first chain. Then I'm adding another column to specify the chain.

2. Assess convergence via traceplots, coloring by the chain.
3. Compute the approximate posterior probability that $\theta < 106$.
4. Obtain a 90% credible interval for `sigma`.

Summarizing coda objects with 'MCMCvis'

In general, I prefer for you to work with the output of JAGS as described above, but many people have created tools to make it easier to work with coda objects. One such tool is the `MCMCvis` package. Go ahead and install the package then load it in:

```
library(MCMCvis)
```

We will look at two functions from this package: `MCMCsummary()` and `MCMCtrace()`.

'MCMCsummary()'

You can obtain a summary of the MCMC chains contained in a coda object by using the `MCMCsummary()` function. You may get a warning if/when you've only run one chain; that's fine! All of the variables in the `variable.names` argument to the `coda.samples()` function will be summarized. Note: these summary quantities are taken over all the chains combined.

```
MCMCsummary(jags_out)
```

```
           mean        sd      2.5%       50%    97.5% Rhat n.eff
sigma2 103.4954 5.7492803  92.83361 103.2963 115.3606    1  5705
theta  106.4117 0.5812041 105.28691 106.4081 107.5537    1  9761
```

You get posterior mean, standard deviation, quantiles, a convergence diagnostic called `Rhat` (don't worry about this yet), and the number of effective samples `n.eff`. The output is a data frame, which you can wrangle or index as you'd like! If you have particular parameter/derived quantity of interest, you can access its specific summary by specifying the `params` argument of `MCMCsummary()`:

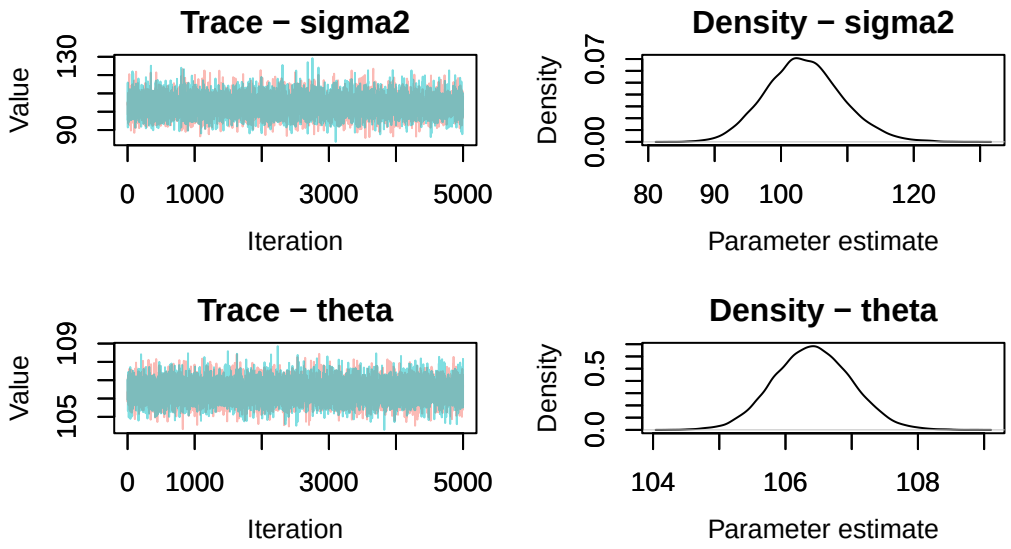
```
MCMCsummary(jags_out, params = "theta")
```

```
          mean        sd     2.5%       50%    97.5% Rhat n.eff
theta 106.4117 0.5812041 105.2869 106.4081 107.5537    1  9761
```

'MCMCtrace()'

We can obtain trace plots and posterior density plots using the `MCMCtrace()` function. By default, this function will create two plots (one traceplot and one density plot) for every parameter. The `pdf = F` just tells `R` to not automatically save the plot as a PDF to your machine. Notice that the two colors in the traceplots represent the two different chains.

```
MCMCtrace(jags_out, pdf = F)
```



Solutions

Checks

Check 1: Complete pooling: we're treating all players as coming from same sampling model!

Check 2: $\theta \sim N(100, 25)$ independent of $\sigma^2 \sim \text{Gamma}(25, 1)$.

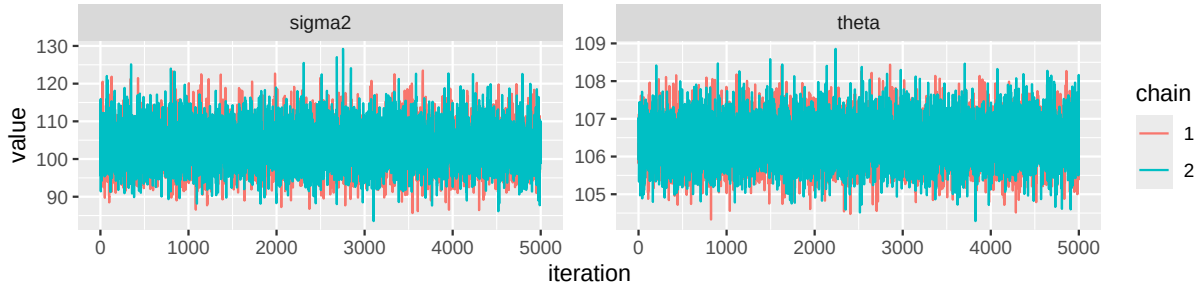
Check 3: Since we have two chains and said we wanted to store 5000 iterations, we should have 10000 samples in total for each parameter/derived quantity.

Practice

```
library(tidyverse)

# Question 1
posts_df <- data.frame(do.call(rbind, jags_out)) |>
  add_column(chain = factor(rep(1:2, each = B)))

# Question 2
posts_df |>
  group_by(chain) |>
  mutate(iteration = row_number()) |>
  ungroup() |>
  pivot_longer(cols = all_of(1:2), names_to = "param", values_to = "value") |>
  ggplot(aes(x = iteration, y = value, col = chain)) +
  geom_line() +
  facet_wrap(~ param, scales = "free")
```




```
## seems like we've converged!

# Question 3
mean(posts_df$theta < 106)
```

```
[1] 0.241
```

```r
# Question 4
quantile(sqrt(posts_df$sigma2), c(0.05, 0.95))
```

```
       5%       95% 
 9.716166 10.645965 
```