(Re)Doing Bayesain Data Analysis

R Pruim

2019-03-27

Contents

Chapter 1

What's in These Notes

This "book" is a companion to Kruschke's *Doing Bayesian Data Analysis*. The main reasons for this companion are to use a different style of R code that includes:

- use of modern packages like `tidyverse`, `R2jags`, `bayesplot`, and `ggformula`;
- adherence to a different style guide;
- less reliance on manually editing scripts and more use of resusable code available in packages;
- a workflow that takes advantage of RStudio and RMarkdown.

This is a work in progress. Please accept my apologies in advance for

- errors,
- inconsistencies
- lack of complete coverage

But feel free to post an issue on github if you spot things that require attention or care to make suggestions for improvement.

I'll be teaching from this book in Spring 2019, so I expect rapid development during those months.

Part I

The Basics: Models, Probability, Bayes, and R

Chapter 2

Credibility, Models, and Parameters

2.1 The Steps of Bayesian Data Analysis

In general, Bayesian analysis of data follows these steps:

1. Identify the **data** relevant to the research questions.

   What are the measurement scales of the data? Which data variables are to be predicted, and which data variables are supposed to act as predictors?

2. Define a **descriptive model for the** relevant **data**. The mathematical form and its parameters should be meaningful and appropriate to the theoretical purposes of the analysis.

3. Specify a **prior distribution** on the parameters. The prior must pass muster with the audience of the analysis, such as skeptical scientists.

4. Use Bayesian inference to **re-allocate credibility across parameter values**. Interpret the posterior distribution with respect to theoretically meaningful issues (assuming that the model is a reasonable description of the data; see next step).

5. Check that the posterior predictions mimic the data with reasonable accuracy (i.e., conduct a "**posterior predictive check**"). If not, then consider a different descriptive model.

In this chapter we will focus on two examples so we can get an overview of what Bayesian data analysis looks like. In subsequent chapters we will fill in lots of the missing details.

2.1.1 R code

Some of the R code used in this chapter has been hidden, and some of it is visible. In any case the point of this chapter is not to understand the details of the R code. It is there mainly for those of you who are curious, or because you might come back and look at this chapter later in the semester.

For those of you new to R, we will be learning it as we go along. For those of you who have used R before, some of this will be familiar to you, but other things likely will not be familiar.

2.1.2 R packages

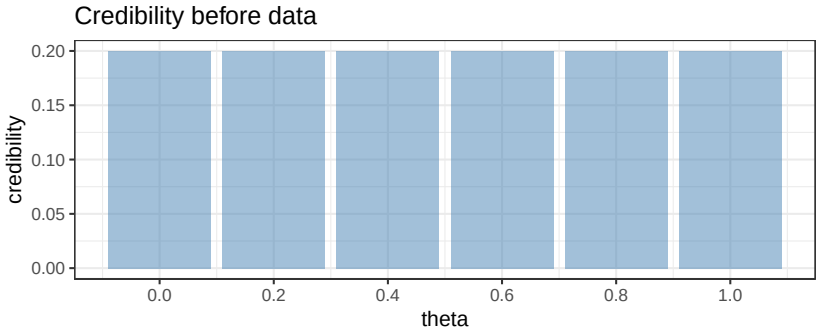
We will make use of a number of R packages as we go along. Here is the code used to load the packages used in this chapter. If you try to mimic the code on your own machine, you will need to use these packages.

```
library(ggformula)        # for creating plots
theme_set(theme_bw())     # change the default graphics settings
library(dplyr)            # for data wrangling
library(mosaic)           # includes the previous 2 (and some other stuff)
library(CalvinBayes)      # includes BernGrid()
library(brms)             # used to fit the model in the second exmample,
                          # but hidden from view here
```

2.2 Example 1: Which coin is it?

As first simple illustration of the big ideas of Bayesian inference, let's consider a situation where we have a coin that is known to result in heads in either 0, 20, 40, 60, 80, or 100% of tosses. But we don't know which. Our plan is to gather data by flipping the coin and recording the results. If we let θ be the true probability of tossing a head, we can refer to these 5 possibilities as $\theta = 0$, $\theta = 0.2$, $\theta = 0.4$, $\theta = 0.6$, $\theta = 0.8$, and $\theta = 1$.

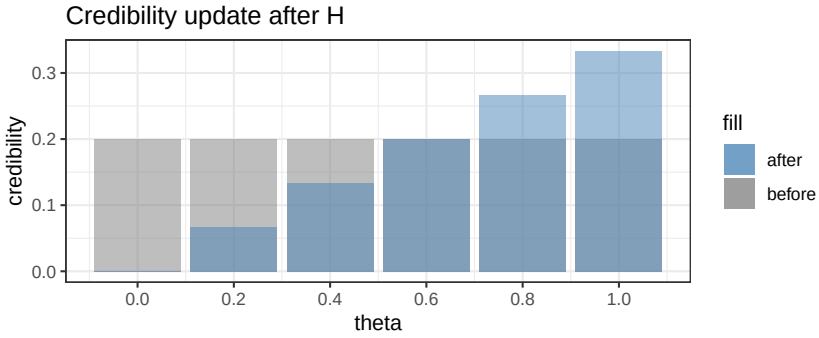
Before collecting our data, if have no other information, we will consider each coin to be equally credible. We could represent that as follows.



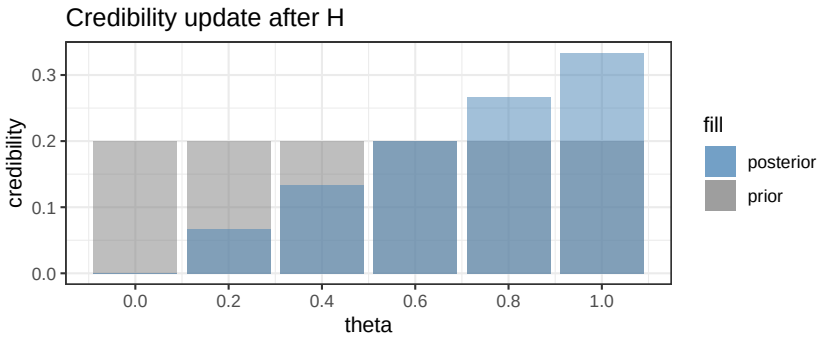
Now suppose we toss the coin and obtain a head. What does that do to our credibilities? Clearly $\theta = 0$ is no longer possible. So the credibility of that option becomes 0. The other credibilities are adjusted as well. We will see later just how, but the following should be intuitive:

- the options with larger values of θ should increase in credibility more than those with lower values of θ.
- the total credibility of all options should remain 1 (100%).

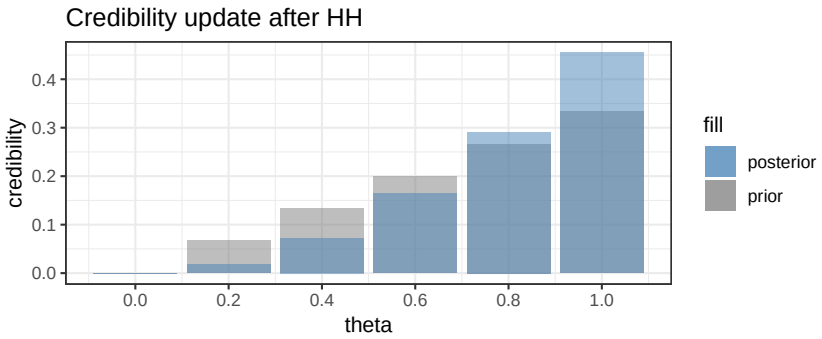
In fact, the adjusted credibility after one head toss looks like this:



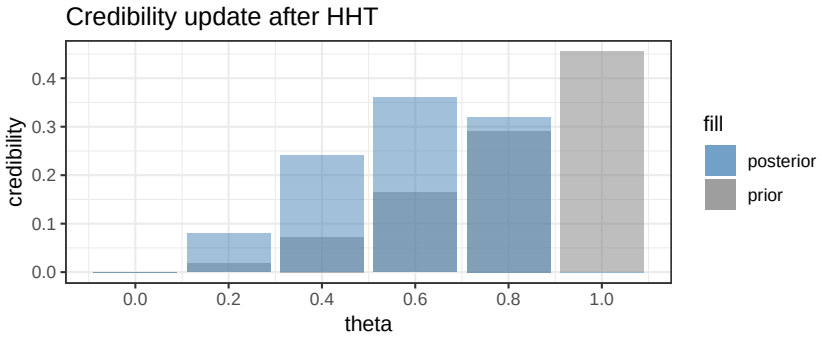
This updating of credibility of possible values of θ is the key idea in Bayesian inference. Bayesians don't call these distributions of credibility "before" and "after", however. Instead they use the longer words "prior" and "posterior", which mean the same thing.



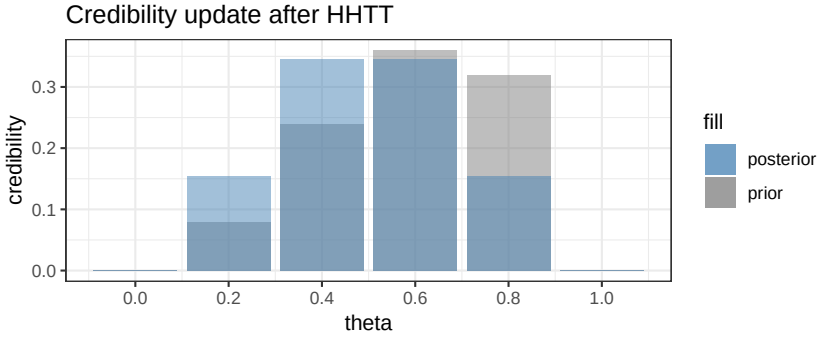
Now suppose we toss the coin again and get another head. Once again we can update the credibility, and once again, the larger values of θ will see their credibility increase while the smaller values of θ will see their credibility decrease.



Time for a third toss. This time we obtain a tail. Now the credibility of $\theta = 1$ drops to 0, and the relative credibilities of the smaller values of θ will increase and of the larger values of θ will decrease.



Finally, we flip one more tail.



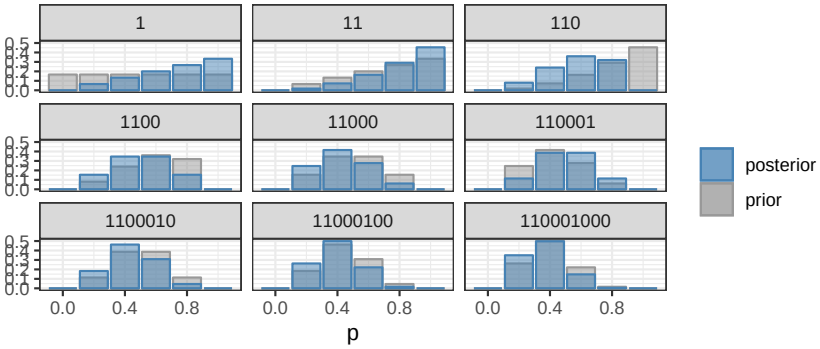
As expected, the posterior is now symmetric with the two central values of θ having the larger credibility.

We can keep playing this game as long as we like. Each coin toss provides a bit more information with which to update the posterior, which becomes our new prior for subsequent data. The `BernGrid()` function in the

`CalvinBayes` package makes it easy to generate plots similar to the ones above. [1]

```
BernGrid("HHTTTHTTT",                          # the data
         steps = TRUE,                         # show each step
         p = c(0, 0.2, 0.4, 0.6, 0.8, 1))      # possible probabilities
```

```
## Converting data to 1, 1, 0, 0, 0, 1, 0, 0, 0
```

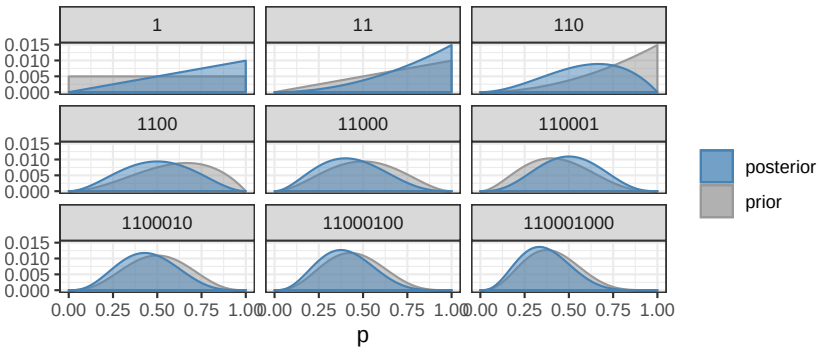


2.2.1 Freedom of choice

In practice, we are usually not given a small number of possible values for the probability (of obtaining heads in our example, but it could be any probability). Instead, the probability could be any value between 0 and 1. But we can do Bayesian updating in essentially the same way. Instead of a bar chart, we will use a line graph (called a density plot) to show how the credibility depends on the parameter value.

```
BernGrid("HHTTTHTTT",                          # the data
         steps = TRUE)                         # show each step
```

```
## Converting data to 1, 1, 0, 0, 0, 1, 0, 0, 0
```



2.3 Distributions

The (prior and posterior) distributions in the previous plots were calculated numerically using a Bayesian update rule that we will soon learn. Density functions have the properties that * they are never negative, and * the total area under the curve is 1. Where the density curve is taller, values are more likely. So in the last posterior credibility above, we see that values near 1/3 are the most credible while values below 0.015 or above 0.065 are not very credible. In particular, we still can't discount the possibility that we are dealing with a fair coin since 0.5 lies well within the most credible central portion of the plot.

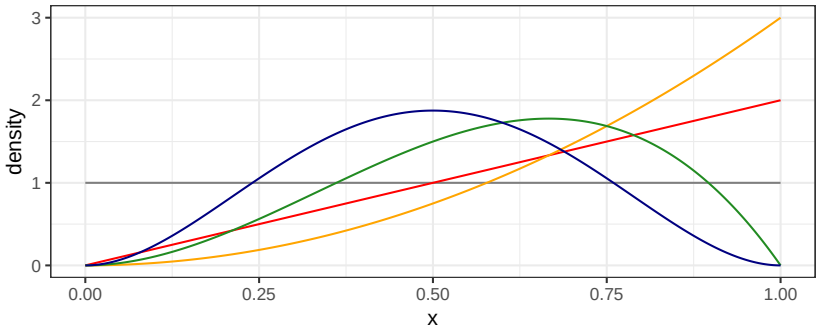
[1] You don't need to know this now, but the name `BernGrid()` comes from the facts that (a) an random process with two outcomes is called a Bernoulli experiment, and (b) the method used to compute the posterior is called the grid method.

We will also encounter densities with names like "normal", "beta", and "t". The `gf_dist()` function from `ggformula` can be used to plot distributions. We just need to provide R's version of the name for the family and any required parameter values.

2.3.1 Beta distributions

The curves in our coins example above look a lot like beta distributions. In fact, we will eventually learn that they are beta distributions, and that each new observed coin toss increases either `shape1` or `shape2` by 1.

```
gf_dist("beta", shape1 = 1, shape2 = 1, color = "gray50") %>%
gf_dist("beta", shape1 = 2, shape2 = 1, color = "red") %>%
gf_dist("beta", shape1 = 3, shape2 = 1, color = "orange") %>%
gf_dist("beta", shape1 = 3, shape2 = 2, color = "forestgreen") %>%
gf_dist("beta", shape1 = 3, shape2 = 3, color = "navy")
```

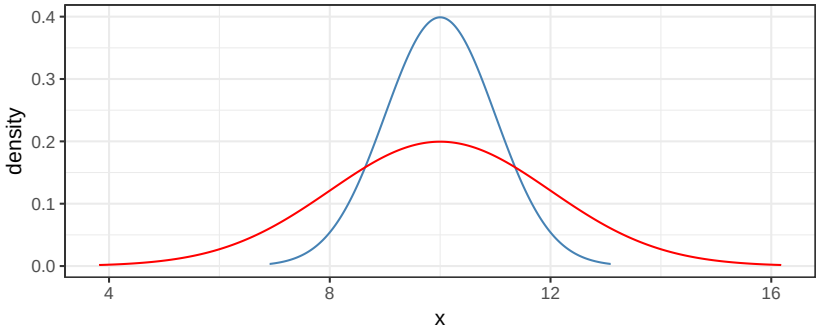


2.3.2 Normal distributions

Another important family of distributions is the normal family. These are bell-shaped, symmetric distributions centered at the mean (μ). A second parameter, the standard deviation (σ) quantifies how spread out the distribution is.

To plot a normal distribution with mean 10 and standard deviation 1 or 2, we use

```
gf_dist("norm", mean = 10, sd = 1, color = "steelblue") %>%
gf_dist("norm", mean = 10, sd = 2, color = "red")
```



The red curve is "twice as spread out" as the blue one.

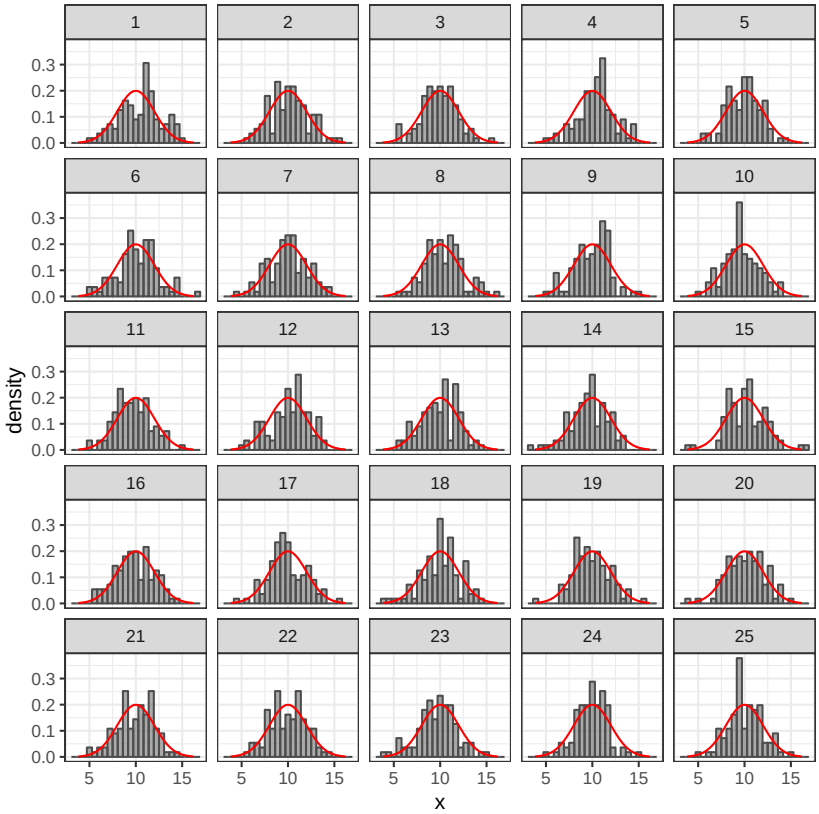
We can also draw random samples from distributions. Random samples will not exactly follow the shape of the distribution they were drawn from, so it takes some experience to get calibrated to know when things are "close enough" to consider a proposed distribution to be believable, and when they are "different enough" to be skeptical. Generating some random data and comparing to the theoretical distribution can help us calibrate.

In the example below, we generate 25 random samples of size 100 and compare their (density) histograms to the theoretical Norm(10, 2) distribution. `expand.grid()` produces a data frame with two columns containing every combination of the numbers 1 through 100 with the numbers 1 through 25, for a total of 2500 rows. `mutate()` is used to add a new variable to the data frame.

```
Rdata <-
  expand.grid(
    rep = 1:100,
    sample = 1:25) %>%
  mutate(
    x = rnorm(2500, mean = 10, sd = 2)
  )
head(Rdata)
```

rep	sample	x
1	1	11.171
2	1	11.419
3	1	9.781
4	1	9.093
5	1	11.212
6	1	6.364

```
gf_dhistogram( ~ x | sample, data = Rdata,
               color = "gray30", alpha = 0.5) %>%
  gf_dist("norm", mean = 10, sd = 2, color = "red")
```



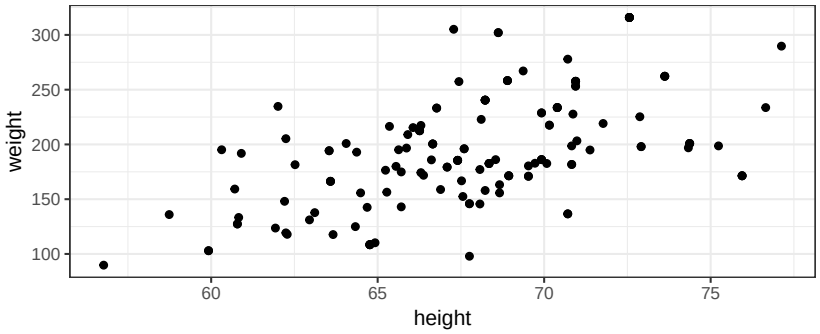
We will see many other uses of these functions. See the next chapter for in introduction to R functions that will be useful.

2.4 Example 2: Height vs Weight

The coins example above is overly simple compared to typical applications. Before getting to the nuts and bolts of doing Bayesian data analysis, let's look at a somewhat more realistic example. Suppose we want to model the relationship between weight and height in 40-year-old Americans.

2.4.1 Data

Here's a scatter plot of some data from the NHANES study that we will use for this example. (Note: this is not the same data set used in the book. The data here come from the `NHANES::NHANES` data set.)



2.4.2 Describing a model for the relationship between height and weight

A plausible model is that weight is linearly related to height. We will make this model a bit more precise by defining the **model parameters** and distributions involved.

Typically statisticians use Greek letters to represent parameters. This model has three parameters (β_0, β_1, and σ) and makes two claims

1. The **average weight** of people with height x is $\beta_0 + \beta_1 x$ (some linear function of x). We can express this as

$$\mu_{Y|x} = E(Y \mid x) = \beta_0 + \beta_1 x$$

or

$$\mu_{\text{weight}|\text{height}} = E(\text{weight} \mid \text{height}) = \beta_0 + \beta_1 \cdot \text{height}$$

The Y and x notation is useful for general formulas; for specific problems (especially in R code), it is usually better to use descriptive names for the variables. The capital Y indicates that it has a distribution. x is lower case because we are imagining a specific value there. So for each value of x, the there is a distribution of Y's.

2. But **not everyone is average**. The model used here assumes that the distributions of the heights of people with a given weight are symmetrically distributed around the average weight for that height and that the distribution is normal (bell-shaped). The parameter σ is called the standard deviation and measures the amount of variability. If σ is small, then most people's weights are very close to the average for their height. If σ is larger, then there is more variability in weights for people who have the same height. We express this as

$$y \mid x \sim \mathsf{Norm}(\mu_{y|x}, \sigma) \tag{2.1}$$

Notice the $\sim$ in this expression. It is read "is distributed as" and describes the distribution (shape) of some quantity.

Putting this all together, and being a little bit sloppy we might write it this way:

$$Y \sim \mathsf{Norm}(\mu, \sigma) \tag{2.2}$$
$$\mu \sim \beta_0 + \beta_1 x \tag{2.3}$$

In this style the dependence of y on x is implicit (via μ's dependence on x) and we save writing $\mid x$ in a few places.

2.4.3 Prior

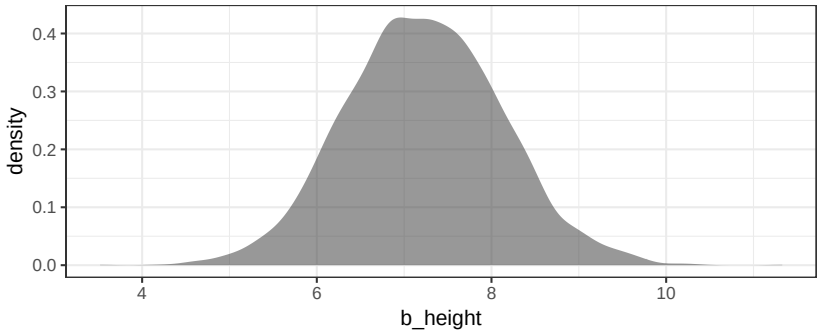
A prior distribution describes what is known/believed about the parameters before we use the information from our data. This could be informed by previous data, or it may be a fairly uninformative prior that considers many values of the parameter to be credible. For this example, we use very flat broad priors (centered at 0 for the β's and extending from 0 to a very large number of σ. (We know that $\sigma > 0$, so our prior should reflect that knowledge.)

2.4.4 Posterior

The posterior distribution is calculated by combining the information about the model (via the **likelihood function**) with the prior. The posterior will provide updated distributions for β_0, β_1 and σ. These distributions will be narrow if our data give a strong evidence about their values and wider if even after considering the data, there is still considerable uncertainty about the parameter values.

For now we won't worry about how the posterior distribution is computed, but we can inspect it visually. (It is called `Post` in the R code below.) For example, if we are primarily interested in the slope (how much heavier are people on average for each inch they are taller?), we can plot the posterior distribution of β_1 or calculate its mean, or the region containing the central 95% of the distribution. Such a region is called a **highest density interval** (HDI) (sometimes called the highest posterior density interval (HPDI), to emphasize that we are looking at a posterior distribution, but an HDI can be computed for other distributions as well).

```
gf_density( ~ b_height, data = Post, alpha = 0.5)
```



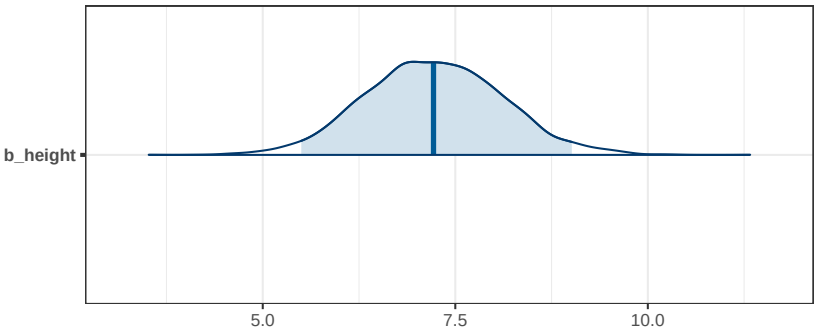

```
mean(~ b_height, data = Post)
```

```
## [1] 7.223
```

```
hdi(Post, pars = "b_height")
```

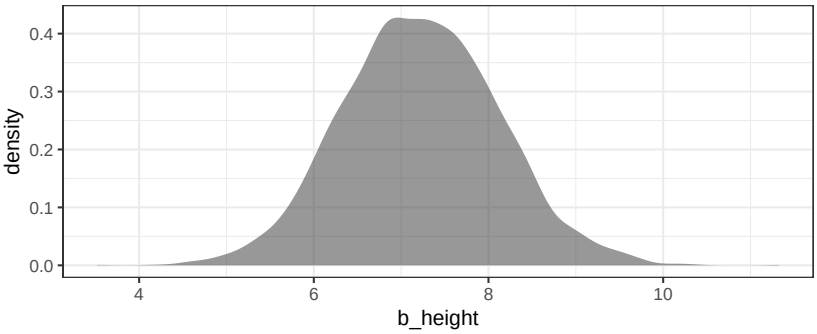
par	lo	hi	prob
b_height	5.551	9.045	0.95

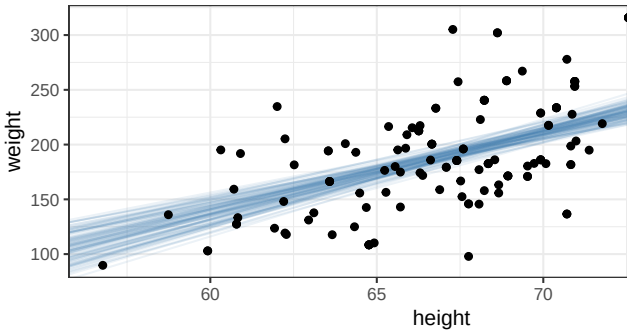
```
mcmc_areas(as.mcmc(Post), pars = "b_height", prob = 0.95)
```



Although we don't get a very precise estimate of β_1 from this model/data combination, we can be quite confident that taller people are indeed heavier (on average), somewhere between 5 and 10 pounds heavier per inch taller.

Another interesting plot shows lines overlaid on the scatter plot. Each line represents a plausible (according to the posterior distribution) combination of slope and intercept. 100 such lines are included in the plot below.





2.4.5 Posterior Predictive Check

Notice that only a few of the dots are covered by the blue lines. That's because the blue lines represent plausible *average* weights. But the model takes into account that some people may be quite a bit heavier or lighter than average. A posterior predictive check is a way of checking that the data look like they could have been plausibly generated by our model.

We can generate a simulated weight for a given height by randomly selecting values of β_0, β_1, σ so that the more credible values are more likely to be selected, and using the normal distribution to generate a difference between an individual weight and the average weight (as determined by the parameters β_0 and β_1. For example, here are the first two rows of our posterior distribution:

```
Post %>% head(2)
```

b_Intercept	b_height	sigma	lp___
-217.0	6.028	40.11	-784.5
-422.1	9.030	48.28	-786.5

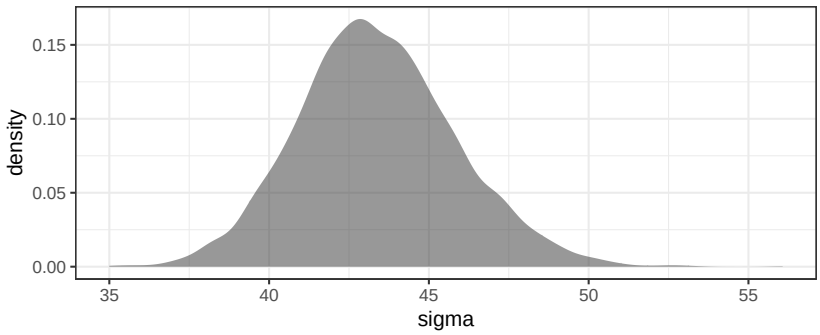
To simulate a weight for a height of 65 inches based on the fist row, we could take a random draw from a $\mathsf{Norm}(-217 + 6.028 \cdot 65, 40.11)$ distribution. We can do a similar thing for the second row.

```
Post %>% head(2) %>%
  mutate(pred_y = rnorm(2, mean = b_Intercept + b_height * 65, sd = sigma))
```

b_Intercept	b_height	sigma	lp___	pred_y
-217.0	6.028	40.11	-784.5	185.3
-422.1	9.030	48.28	-786.5	219.4

Those values are quite different. This is because credible values of σ are quite large – an indication that individuals will vary quite substantially from the average weight for their height.

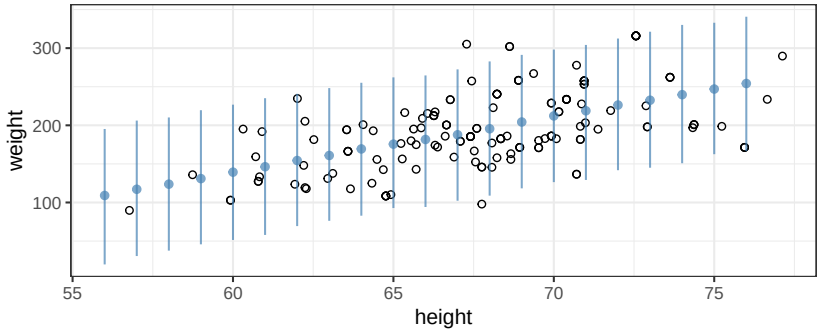
```
gf_density( ~ sigma, data = Post)
```



We should not be surprised to see some ($\sim 5\%$) of people who are 85 pounds above or below the average weight for their height.

If we do this many times for several height values and plot the central 95% of the weights, we get a plot that looks like this:

```
PPC <-
  expand.grid(
    height = seq(56, 76, by = 1),
    rep = 1:nrow(Post)
  ) %>%
  mutate(
    b_Intercept = Post$b_Intercept[rep],
    b_height = Post$b_height[rep],
    sigma = Post$sigma[rep],
    weight = b_Intercept + b_height * height + rnorm(n = n(), 0, sigma)
  ) %>%
  group_by(height) %>%
  summarise(
    mean = mean(weight),
    lo = quantile(weight, prob = 0.025),
    hi = quantile(weight, prob = 0.975)
  )

gf_point(weight ~ height, data = NHANES40, shape = 1) %>%
  gf_pointrange(mean + lo + hi ~ height, data = PPC, alpha = 0.7, color = "steelblue")
```



Now we see that indeed, most (but not all) of the data points fall within a range that the model believes is credible. If this were not the case, it would be evidence that our model is not well aligned with the data and might lead us to explore other models.

Notice that by taking many different credible values of the parameters (including σ), we are taking into account both our uncertainty about the parameter values and the variability that the model describes in the population (even for given parameter values).

2.5 Where do we go from here?

Now that we have seen an overview of Bayesian inference at work, you probably have lots of questions. Of time we will improve our answers to each of them.

1. How do we create models?

   One of the nice things about Bayesian inference is that it is so flexible. That allows us to create all sorts of models. We will begin with models of a proportion (and how that proportion might depend on other variables) because these are the simplest to understand. Then we will move on other important examples. (The back half of our book is a smorgasbord of example situations.)

2. How do we select priors?

   We will begin with fairly "uninformative" priors that say very little, and we will experiment with different priors to see what affect the choice of prior has on our analysis. Gradually we will learn more about prior selection.

3. How do we update the prior based on data to get the posterior?

   Here we will learn several approaches, most of them computational. (There are only a limited number of examples where the prior can be computed analytically.) We will start with computational methods that are simple to implement and relatively easy to understand, but are too inefficient to use on large or complex problems. Eventually we will learn how to use two important algorithms (JAGS and Stan) to describe and fit Bayesian models.

4. How do we tell whether the algorithm that generated the posterior worked well?

   The computainal algorithms that compute posterior distributions can fail. No one algorithm works best on every problem, and sometimes we need to describe our model differently to help the computer. We will learn some diagnostics to help us detect when there may be problems with our computations.

5. What can we do with the posterior once we have it?

   After all the work of building a model, selectig a prior, fitting the model to obtain a posterior, and convincing ourselves that no disasters have happened along the way, what can we do with the posterior? We will use it both to diagnose the model itself and to see what the model has to say.

2.6 Exercises

1. Consider Figure 2.6 on page 29 of *DBDA2E*. Two of the data points fall above the vertical bars. Does this mean that the model does not describe the data well? Briefly explain your answer.

2. Run the following examples in R. Compare the plots produced and comment the big idea(s) illustrated by this comparison.

```
library(CalvinBayes)
BernGrid("H", resolution = 4,  prior = triangle::dtriangle)
BernGrid("H", resolution = 10, prior = triangle::dtriangle)
BernGrid("H", prior = 1, resolution = 100, geom = geom_col)
BernGrid("H", resolution = 100,
         prior = function(p) abs(p - 0.5) > 0.48, geom = geom_col)
```

3. Run the following examples in R. Compare the plots produced and comment the big idea(s) illustrated by this comparison.

```
library(CalvinBayes)
BernGrid("TTHT", prior = triangle::dtriangle)
BernGrid("TTHT",
         prior = function(x) triangle::dtriangle(x)^0.1)
BernGrid("TTHT",
         prior = function(x) triangle::dtriangle(x)^10)
```

4. Run the following examples in R. Compare the plots produced and comment the big idea(s) illustrated by this comparison.

```
library(CalvinBayes)
dfoo <- function(p) {
  0.02 * dunif(p) +
  0.49 * triangle::dtriangle(p, 0.1, 0.2) +
  0.49 * triangle::dtriangle(p, 0.8, 0.9)
}
BernGrid(c(rep(0,13), rep(1,14)), prior = triangle::dtriangle)
BernGrid(c(rep(0,13), rep(1,14)), resolution = 1000, prior = dfoo)
```

5. Run the following examples in R. Compare the plots produced and comment the big idea(s) illustrated by this comparison.

```
library(CalvinBayes)
dfoo <- function(p) {
  0.02 * dunif(p) +
  0.49 * triangle::dtriangle(p, 0.1, 0.2) +
  0.49 * triangle::dtriangle(p, 0.8, 0.9)
}
BernGrid(c(rep(0, 3), rep(1, 3)), prior = dfoo)
BernGrid(c(rep(0, 10), rep(1, 10)),  prior = dfoo)
BernGrid(c(rep(0, 30), rep(1, 30)),  prior = dfoo)
BernGrid(c(rep(0, 100), rep(1, 100)), prior = dfoo)
```

6. Run the following examples in R and compare them to the ones in the previous exercise. What do you observe?

```
library(CalvinBayes)
dfoo <- function(p) {
  0.02 * dunif(p) +
  0.49 * triangle::dtriangle(p, 0.1, 0.2) +
  0.49 * triangle::dtriangle(p, 0.8, 0.9)
}
BernGrid(c(rep(0, 3), rep(1, 4)), prior = dfoo)
BernGrid(c(rep(0, 4), rep(1, 3)), prior = dfoo)
BernGrid(c(rep(0, 10), rep(1, 11)),  prior = dfoo)
BernGrid(c(rep(0, 11), rep(1, 10)),  prior = dfoo)
BernGrid(c(rep(0, 30), rep(1, 31)),  prior = dfoo)
BernGrid(c(rep(0, 31), rep(1, 30)),  prior = dfoo)
```

2.7 Footnotes

Chapter 3

Some Useful Bits of R

3.1 You Gotta Have Style

Good programming style is incredibly important. It makes your code easier to read and edit. That leads to fewer errors.

Here is a brief style guide you are expected to follow for all code in this course: For more detailed see http://adv-r.had.co.nz/Style.html, on which this is based.

1. **No long lines.**

   Lines of code should have at most 80 characters.

   Programming lines should not wrap, you should choose the line breaks yourself. Choose them in natural places. In R markdown, long codes lines don't wrap, they flow off the page, so the end isn't visible. (And it makes it obvious that you didn't look at your own print out.)

```
# Don't ever use really long lines.  Not even in comments.  They spill off the page and make people
```

2. **Use your space bar.**

   There is a reason it is the largest key on the keyboard. Use it often. Spaces **after commas** (always). Spaces **around operators** (always). Spaces after the comment symbol **#** (always). Use other spaces judiciously to align similar code chunks to make things easier to read or compare.

```
x<-c(1,2,4)+5          # BAD BAD BAD
x <- c(1, 2, 4) + 5    # Ah :^)
```

3. But don't go crazy with the space bar.

   There are a few places you should not use spaces:

   - after open parentheses or before closed parentheses
   - between function names and the parentheses that follow

4. **Indent** to show the structure of your code.

   Use 2 spaces to indent (to keep things from drifting right too quickly).

   Fortunately, this is really easy. Highlight your code and hit `<CTRL>`-i (PC) or `<command>`-i (Mac). If the indention looks odd to you, you have most likely messed up commas, quotes, parentheses, or curly braces.

5. **Choose names wisely and consistently**.

   Naming things is hard, but take a moment to choose good names, and go back and change them if you come up with a better name later. Here are some helpful hints:

a. Very **short names** should only be used for a very **short time** (a couple lines of code). Else we tend to forget what they meant. Avoid names like `x`, `f`, etc. unless the use is brief and mimics some common mathematical formula.

b. **Break long names visually**. Common ways to do this are with a dot (`.`), an underscore `_`, or `camelCase`. There are R coders who prefer all three, but don't mix and match for similar kinds of things, that just makes it harder to remember what to do the next time.

```
good_name <- 10
good.name <- 10
goodName  <- 10    # note alignment via extra space in this line
really_terrible.ideaToDo <- -5
```

The trend in R is toward using underscore (`_`) and I recommend it. Older code often used dot (`.`). CamelCase is the least common in R.

c. Recommendation: capitalize data frame names; use lower case for variables inside data frames.

This is not a common convention in R, but it can really help to keep things straight. I'll do this in the data sets I create, but when we use other data sets, they may not follow this convention.

d. Avoid using names that are already in use by R.

This can be hard to avoid when you are starting out because you don't know what all is defined. Here are a few things to avoid.

```
T           # abbreviation for TRUE
F           # abbreviation for FALSE
c           # used to concetenate vectors and lists
df          # density function for f distributions
dt          # density function for t distributions
```

6. **Use comments (`#`)**, but use them for the right thing.

Comments can be used to clarify names, point out subtlties in code, etc. They should not be used for your analysis or discussion of results. Don't comment things that are obvious without comment. Comments should add value.

```
x <- 4   # set x to 4  <---------------------- no need for this comment
x <- 4   # b/c there are four grade levels in the study  <------- useful
```

7. **Exceptions** should be exceptional.

No style guide works perfectly in all situations. Ocassionally you may need to violate the style guide. But these instances should be rare and should have a good reason. They should not arise form your sloppiness or laziness.

3.1.1 An additional note about homework

When you do homework, I want to see your code and the results (and your discussion of those results). Writing in R Markdown makes this all easy to do.

But make sure that I can see all the necessary things to evaluate what you are doing. You have access to your code and can investigate variables, etc. But make sure I can see what's going one in the document. This often means displaying intermediate results. Once common way to do this is with a semi-colon:

```
x <- 57 * 23; x
```

```
## [1] 1311
```

3.2 Vectors, Lists, and Data Frames

3.2.1 Vectors

In R, a vector is a homogeneous ordered collection (indexing starts at 1). By homogeneous, we mean that each element is he same kind of thing.

Short vectors can be created using `c()`:

```
x <- c(1, 3, 5)
x
```

```
## [1] 1 3 5
```

Evenly spaced sequences can be created using `seq()`:

```
x <- seq(0, 100, by = 5); x
```

```
##  [1]   0   5  10  15  20  25  30  35  40  45  50  55  60  65  70  75  80
## [18]  85  90  95 100
```

```
y <- seq(0, 1, length.out = 11); y
```

```
##  [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

```
0:10      # short cut for consecutive integers
```

```
##  [1]  0  1  2  3  4  5  6  7  8  9 10
```

Repeated values can be created with `rep()`:

```
rep(5, 3)
```

```
## [1] 5 5 5
```

```
rep(c(1, 2, 3), each  = 2)
```

```
## [1] 1 1 2 2 3 3
```

```
rep(c(1, 2, 3), times = 2)
```

```
## [1] 1 2 3 1 2 3
```

```
rep(c(1, 2, 3), times = c(1, 2, 3))
```

```
## [1] 1 2 2 3 3 3
```

```
rep(c(1, 2, 3), each  = c(1, 2, 3))    # Ack! see warning message.
```

```
## Warning in rep(c(1, 2, 3), each = c(1, 2, 3)): first element used of 'each'
## argument
```

```
## [1] 1 2 3
```

When a function acts on a vector, there are several things that could happen.

- One result can be computed from the entire vector.

  ```
  x <- c(1, 3, 6, 10)
  length(x)
  ```

  ```
  ## [1] 4
  ```

  ```
  mean(x)
  ```

  ```
  ## [1] 5
  ```

- The function may be applied to each element of the vector and a vector of results returned. (Such functions are called **vectorized**.)

```
log(x)
```

```
## [1] 0.000 1.099 1.792 2.303
```

```
2 * x
```

```
## [1]  2  6 12 20
```

```
x^2
```

```
## [1]   1   9  36 100
```

- The first element of the vector may be used and the others ignored. (Less common but dangerous – be on the lookout. See example above.)

Items in a vector can be accessed using `[]`:

```
x <- seq(10, 20, by = 2)
x[2]
```

```
## [1] 12
```

```
x[10]       # NA indicates a missing value
```

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

```
x[10] <- 4
x           # missing values filled in to make room!
```

```
##  [1] 10 12 14 16 18 20 NA NA NA  4
```

```
is.na(x)
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE FALSE
```

In addition to using integer indices, there are two other ways to access elements of a vector: names and logicals. If the items in a vector are named, names are displayed when a vector is displayed, and names can be used to access elements.

```
x <- c(a = 5, b = 3, c = 12, 17, 1)
x
```

```
##  a  b  c
##  5  3 12 17  1
```

```
names(x)
```

```
## [1] "a" "b" "c" ""  ""
```

```
x["b"]
```

```
## b
## 3
```

Logicals (`TRUE` and `FALSE`) are very interesting in R. In indexing, they tell us which items to keep and which to discard.

```
x <- (1:10)^2
x < 50
```

```
##  [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE
```

```
x[x < 50]
```

```
## [1]  1  4  9 16 25 36 49
```

```
x[c(TRUE, FALSE)]   # T/F recycled to length 10
```

```
## [1]  1  9 25 49 81
```

```
which(x < 50)
```

```
## [1] 1 2 3 4 5 6 7
```

3.2.2 Lists

Lists are a lot like vectors, but

- Create a list with `list()`.
- The elements can be different kinds of things (including other lists)
- Use `[[ ]]` to access elements. You can also use `$` to access named elements
- If you use `[]` you will get a list back not an element.

```
# a messy list
L <- list(5, list(1, 2, 3), x = TRUE, y = list(5, a = 3, 7))
L
```

```
## [[1]]
## [1] 5
##
## [[2]]
## [[2]][[1]]
## [1] 1
##
## [[2]][[2]]
## [1] 2
##
## [[2]][[3]]
## [1] 3
##
##
## $x
## [1] TRUE
##
## $y
## $y[[1]]
## [1] 5
##
## $y$a
## [1] 3
##
## $y[[3]]
## [1] 7
```

```
L[[1]]
```

```
## [1] 5
```

```
L[[2]]
```

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

```
##
## [[2]]
## [1] 2
##
## [[3]]
## [1] 3
```

```
L[1]
```

```
## [[1]]
## [1] 5
```

```
L$y
```

```
## [[1]]
## [1] 5
##
## $a
## [1] 3
##
## [[3]]
## [1] 7
```

```
L[["x"]]
```

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

```
L[1:3]
```

```
## [[1]]
## [1] 5
##
## [[2]]
## [[2]][[1]]
## [1] 1
##
## [[2]][[2]]
## [1] 2
##
## [[2]][[3]]
## [1] 3
##
##
## $x
## [1] TRUE
```

```
glimpse(L)
```

```
## List of 4
##  $  : num 5
##  $  :List of 3
##   ..$ : num 1
##   ..$ : num 2
##   ..$ : num 3
##  $ x: logi TRUE
##  $ y:List of 3
##   ..$  : num 5
##   ..$ a: num 3
```

```
##   ..$  : num 7
```

3.2.3 Data frames for rectangular data

Rectangular data is organized in rows and columns (much like an excel spreadsheet). These rows and columns have a particular meaning:

- Each **row** represents one **observational unit**. Observational units go by many others names depending whether they are people, or inanimate objects, our events, etc. Examples include case, subject, item, etc. Regardless, the observational units are the things about which we collect information, and each one gets its own row in rectangular data.
- Each **column** represents a **variable** – one thing that is "measured" and recorded (at least in principle, some measurements might be missing) for each observational unit.

Example In a study of nutritional habits of college students, our observational units are the college students in the study. Each student gets her own row in the data frame. The variables might include things like an ID number (or name), sex, height, weight, whether the student lives on campus or off, what type of meal plan they have at the dining hall, etc., etc. Each of these is recorded in a separate column.

Data frames are the standard way to store **rectangular data** in R. Usually variables (elements of the list) are vectors, but this isn't required, sometimes you will see list variables in data frames. Each element (ie, variable) must have the same length (to keep things rectangular).

Here, for example, are the first few rows of a data set called `KidsFeet`:

```
library(mosaicData)   # Load package to make KidsFeet data available
head(KidsFeet)        # first few rows
```

name	birthmonth	birthyear	length	width	sex	biggerfoot	domhand
David	5	88	24.4	8.4	B	L	R
Lars	10	87	25.4	8.8	B	L	L
Zach	12	87	24.5	9.7	B	R	R
Josh	1	88	25.2	9.8	B	L	R
Lang	2	88	25.1	8.9	B	L	R
Scotty	3	88	25.7	9.7	B	R	R

3.2.3.1 Accessing via [ ]

We can access rows, columns, or individual elements of a data frame using `[ ]`. This is the more usual way to do things.

```
KidsFeet[, "length"]
```

```
##  [1] 24.4 25.4 24.5 25.2 25.1 25.7 26.1 23.0 23.6 22.9 27.5 24.8 26.1 27.0
## [15] 26.0 23.7 24.0 24.7 26.7 25.5 24.0 24.4 24.0 24.5 24.2 27.1 26.1 25.5
## [29] 24.2 23.9 24.0 22.5 24.5 23.6 24.7 22.9 26.0 21.6 24.6
```

```
KidsFeet[, 4]
```

```
##  [1] 24.4 25.4 24.5 25.2 25.1 25.7 26.1 23.0 23.6 22.9 27.5 24.8 26.1 27.0
## [15] 26.0 23.7 24.0 24.7 26.7 25.5 24.0 24.4 24.0 24.5 24.2 27.1 26.1 25.5
## [29] 24.2 23.9 24.0 22.5 24.5 23.6 24.7 22.9 26.0 21.6 24.6
```

```
KidsFeet[3, ]
```

	name	birthmonth	birthyear	length	width	sex	biggerfoot	domhand
3	Zach	12	87	24.5	9.7	B	R	R

```
KidsFeet[3, 4]
```

```
## [1] 24.5
```

```
KidsFeet[3, 4, drop = FALSE]  # keep it a data frame
```

	length
3	24.5

```
KidsFeet[1:3, 2:3]
```

birthmonth	birthyear
5	88
10	87
12	87

By default,

- Accessing a row returns a 1-row data frame.
- Accessing a column returns a vector (at least for vector columns)
- Accessing a element returns that element (technically a vector with one element in it).

3.2.3.2 Accessing columns via $

We can also access individual variables using the `$` operator:

```
KidsFeet$length
```

```
##  [1] 24.4 25.4 24.5 25.2 25.1 25.7 26.1 23.0 23.6 22.9 27.5 24.8 26.1 27.0
## [15] 26.0 23.7 24.0 24.7 26.7 25.5 24.0 24.4 24.0 24.5 24.2 27.1 26.1 25.5
## [29] 24.2 23.9 24.0 22.5 24.5 23.6 24.7 22.9 26.0 21.6 24.6
```

```
KidsFeet$length[2]
```

```
## [1] 25.4
```

```
KidsFeet[2, "length"]
```

```
## [1] 25.4
```

```
KidsFeet[2, "length", drop = FALSE]    # keep it a data frame
```

	length
2	25.4

As we will see, there are are other tools that will help us avoid needing to us `$` or `[ ]` for access columns in a data frame. This is especially nice when we are working with several variables all coming from the same data frame.

3.2.3.3 Accessing by number is dangerous

Generally speaking, it is safer to access things by name than by number when that is an option. It is easy to miscalculate the row or column number you need, and if rows or columns are added to or deleted from a data frame, the numbering can change.

3.2.3.4 Implementation

Data frames are implemented in R as a special type (technically, class) of list. The elements of the list are the *columns* in the data frame. Each column must have the same length (so that our data frame has coherent *rows*). Most often the columns are vectors, but this isn't required.

This explains why `$` works the way it does – we are just accessing one item in a list. It also means that we can use `[[ ]]` to access a column:

```
KidsFeet[["length"]]
```

```
##  [1] 24.4 25.4 24.5 25.2 25.1 25.7 26.1 23.0 23.6 22.9 27.5 24.8 26.1 27.0
## [15] 26.0 23.7 24.0 24.7 26.7 25.5 24.0 24.4 24.0 24.5 24.2 27.1 26.1 25.5
## [29] 24.2 23.9 24.0 22.5 24.5 23.6 24.7 22.9 26.0 21.6 24.6
```

```
KidsFeet[[4]]
```

```
##  [1] 24.4 25.4 24.5 25.2 25.1 25.7 26.1 23.0 23.6 22.9 27.5 24.8 26.1 27.0
## [15] 26.0 23.7 24.0 24.7 26.7 25.5 24.0 24.4 24.0 24.5 24.2 27.1 26.1 25.5
## [29] 24.2 23.9 24.0 22.5 24.5 23.6 24.7 22.9 26.0 21.6 24.6
```

```
KidsFeet[4]
```

length
24.4
25.4
24.5
25.2
25.1
25.7
26.1
23.0
23.6
22.9
27.5
24.8
26.1
27.0
26.0
23.7
24.0
24.7
26.7
25.5
24.0
24.4
24.0
24.5
24.2
27.1
26.1
25.5
24.2
23.9
24.0
22.5
24.5
23.6
24.7
22.9
26.0
21.6
24.6

3.2.4 Other types of data

Some types of data do not work well in a rectangular arrangement of a data frame, and there are many other ways to store data. In R, other types of data commonly get stored in a list of some sort.

3.3 Plotting with ggformula

R has several plotting systems. Base graphics is the oldest. `lattice` and `ggplot2` are both built on a system called grid graphics. `ggformula` is built on `ggplot2` to make it easier to use and to bring in some of the advantages of `lattice`.

You can find out about more about `ggformula` at https://projectmosaic.github.io/ggformula/news/index.html.

3.4 Creating data with expand.grid()

We will frequently have need of synthetic data that includes all combinations of some variable values. `expand.grid()` does this for us:

```
expand.grid(
  a = 1:3,
  b = c("A", "B", "C", "D"))
```

a	b
1	A
2	A
3	A
1	B
2	B
3	B
1	C
2	C
3	C
1	D
2	D
3	D

3.5 Transforming and summarizing data dplyr and tidyr

See the tutorial at http://rsconnect.calvin.edu/wrangling-jmm2019 or https://rpruim.shinyapps.io/wrangling-jmm2019

3.6 Writing Functions

3.6.1 Why write functions?

There are two main reasons for writing functions.

1. You may want to use a tool that requires a function as input.

   To use `integrate()`, for example, you must provide the integrand as a function.

2. To make your own work easier.

   Functions make it easier to reuse code or to break larger tasks into smaller parts.

3.6.2 Function parts

Functions consist of several parts. Most importantly

1. An argument list.

A list of named inputs to the function. These may have default values (or not). There is also a special argument `...` which gathers up any other arguments provided by the user. Many R functions make use of `...`

```
args(ilogit)  # one argument, called x, no default value
```

```
## function (x)
## NULL
```

2. The body.

   This is the code the tells R to do when the function is executed.

```
body(ilogit)
```

```
## {
##     exp(x)/(1 + exp(x))
## }
```

3. An environment where code is executed.

   Each function has its own "scratch pad" where it can do work without interfering with global computations. But environments in R are nested, so it is possible to reach outside of this narrow environment to access other things (and possible to change them). For the most part we won't worry about this, but if you use a variable not defined in your function but defined elsewhere, you may see unexpecte results.

If you type the name of a function without parenthesis, you will see all three parts listed:

```
ilogit
```

```
## function (x)
## {
##   exp(x)/(1 + exp(x))
## }
## <bytecode: 0x7ff653f79c58>
## <environment: namespace:mosaicCore>
```

3.6.3 The function() function has its function

To write a function we use the `function()` function to specify the arguments and the body. (R will assign an environment for us.)

The general outline is

```
my_function_name <-
  function(arg1 = default1, arg2 = default2, arg3, arg4, ...) {

    # stuff for my function to do
  }
```

We may include as many named arguments as we like, and some or all or none of them may have default values. The results of the last line of the function are returned. If we like, we can also use the `return()` function to make it clear what is being returned when.

Let's write a function that adds. (Redundant, but a useful illustration.)

```
foo <- function(x, y = 5) {
  x + y     # or return(x + y)
}
foo(3, 5)
```

```
## [1] 8
```

```
foo(3)
```

```
## [1] 8
```

```
foo(2, x = 3)   # Note: this makes y = 2
```

```
## [1] 5
```

```
foo(x = 1:3, y = 100)
```

```
## [1] 101 102 103
```

```
foo(x = 1:3, y = c(100, 200, 300))  # vectorized!
```

```
## [1] 101 202 303
```

```
foo(x = 1:3, y = c(100, 200))       # You have been warned!
```

```
## Warning in x + y: longer object length is not a multiple of shorter object
## length
```

```
## [1] 101 202 103
```

Here is a more useful example. Suppose we want to integrate $f(x) = x^2(2 - x)$ on the interval from 0 to 2. Since this is such a simple function, if we are not going to reuse it, we don't need to bother naming it, we can just create the function inside our call to `integrate()`.

```
integrate(function(x) { x^2 * (2-x) }, 0, 2)
```

```
## 1.333 with absolute error < 1.5e-14
```

3.7 Some common error messages

3.7.1 object not found

If R claims some object is not found, the two most likely causes are

- a typo – if you spell the name of the object slightly differently, R can't figure out what you mean

  ```
  blah <- 17
  bla
  ```

  ```
  ## Error in eval(expr, envir, enclos): object 'bla' not found
  ```

- forgetting to load a package – if the object is in a package, that package must be loaded

  ```
  detach("package:mosaic", unload = TRUE)
  detach("package:ggformula", unload = TRUE)   # without packages, no gf_line()
  ```

  ```
  ## Error: package 'ggformula' is required by 'CalvinBayes' so will not be detached
  ```

  ```
  gf_line()
  ```

  ```
  ## gf_line() uses
  ##     * a formula with shape y ~ x.
  ##     * geom:  line
  ##     * key attributes:  alpha, color, fill, group, linetype, size, lineend,
  ##                    linejoin, linemitre, arrow
  ##
  ## For more information, try ?gf_line
  ```

```
library(ggformula)      # reload package; now things work
gf_line()
```

```
## gf_line() uses
##     * a formula with shape y ~ x.
##     * geom:  line
##     * key attributes:  alpha, color, fill, group, linetype, size, lineend,
##                     linejoin, linemitre, arrow
##
## For more information, try ?gf_line
```

3.7.2 Package inputenc Error: Unicode char not set up for use with LaTeX.

Sometimes if you copy and paste text from a web page or PDF document you will get symbols knitr doesn't know how to handle. Smart quotes, ligatures, and other special characters are the most likely cause.

`tools::showNonASCIIfile()` can help you locate non-ASCII characters in a file.

3.7.3 Any message mentioning yaml

YAML stand for yet another markup language. The first part of an R Markdown file is called the YAML header. If you get a yaml error when you knit a document, most likely you have messed up the YAML header someone. If you don't see the problem, you can start a new document and copy-and-paste the contents (without the TAML header) into the new document (after its YAML header).

3.8 Exercises

1. Create a function in R that converts Fahrenheit temperatures to Celsius temperatures. [Hint: $C = (F - 32) \cdot 5/9$.]

   What you turn in should show

   a. the code that defines your function.
   b. some test cases that show that your function is working. (Show that -40, 32, 98.6, and 212 convert to -40, 0, 37, and 100.) Note: you should be able to test all these cases by calling the function only once. Use `c(-40, 32, 98.6, 212)` as the input.

2. See if you can predict the output of each line below. Then run in R to see if you are correct. If you are not correct, see if you can figure out why R does what it does (and make a note so you are not surprised the next time).

```
odds <- 1 + 2 * (0:4); odds
primes <- c(2, 3, 5, 7, 11, 13)
length(odds)
length(primes)
odds + 1
odds + primes
odds * primes
odds > 5
sum(odds > 5)
sum(primes < 5 | primes > 9)
odds[3]
odds[10]
odds[-3]
primes[odds]
```

```
primes[primes >= 7]
sum(primes[primes > 5])
sum(odds[odds > 5])
odds[10] <- 1 + 2 * 9
odds
y <- 1:10
y <- 1:10; y
(x <- 1:5)
```

3. The problem uses the `KidsFeet` data set from the `mosaicData` package. The hints are suggested functions that might be of use.
   a. How many kids are represented in the data set. [Hint: `nrow()` or `dim()`]
   b. Which of the variables are factors? [Hint: `glimpse()`]
   c. Add a new variable called `foot_ratio` that is equal to `length` divided by `width`. [Hint: `mutate()`]
   d. Add a new variable called `biggerfoot2` that has values `"dom"` (if `domhand` and `biggerfoot` are the same) and `"nondom"` (if `domhand` and `biggerfoot` are different). [Hint: `mutate()`, `==`, `ifelse()`]
   e. Create new data set called `Boys` that contains only the boys. [Hint: `filter()`, `==`]
   f. What is the name of the boy with the largest `foot_ratio`? Show how to find this programmatically, don't just scan through the whole data set yourself. [Hint: `max()` or `arrange()`]

3.9 Footnotes

Chapter 4

Probability

4.1 Some terminology

Probability is about quantifying the relative chances of various possible outcomes of a random process.

As a very simple example (used to illustrate the terminology below), considering rolling a single 6-sided die.

sample space: The set of all possible outcomes of a random process. [{1, 2, 3, 4, 5, 6}]

event: a set of outcomes (subset of sample space) [E = {2, 4, 6} is the event that we obtain an even number]

probability: a number between 0 and 1 assigned to an event (really a function that assigns numbers to each event). We write this P(E). [P(E) = 1/2 where E = {1, 2, 3}]

random variable: a random process that produces a number. [So rolling a die can be considered a random variable.]

probability distribution: a description of all possible outcomes and their probabilities. For rolling a die we might do this with a table like this:

1	2	3	4	5	6
1/6	1/6	1/6	1/6	1/6	1/6

support (of a random variable): the set of possible values of a random variable. This is very similar to the sample space.

probability mass function (pmf): a function (often denoted with p or f) that takes possible values of a discrete random variable as input and returns the probability of that outcome.

- If S is the support of the random variable, then

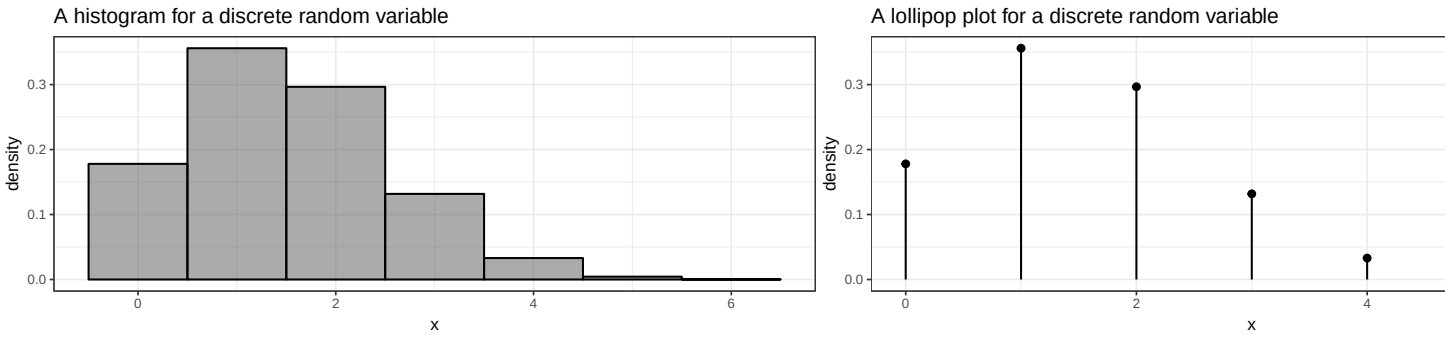
$$\sum_{x \in S} p(x) = 1$$

  and any function with this property is a pmf.

- Probabilities of events are obtained by adding the probabilities of all outcomes in the event:

$$\Pr(E) = \sum_{x \in E} p(x)$$

* pmfs can be represented in a table (like the one above) or graphically with a probability histogram or lollipop plot like the ones below. [These are not for the 6-sided die, as we can tell because the probabilities are not the same for each input; the die rolling example would make very boring plots.]



- Histograms are generally presented on the **density scale** so the total area of the histogram is 1. (In this example, the bin widths are 1, so this is the same as being on the probability scale.)

probability density function (pdf): a function (often denoted with p or f) that takes the possible values of continuous random variable as input and returns the probability *density*.

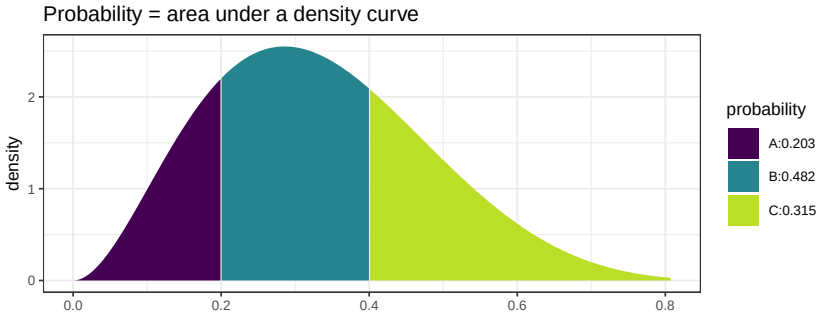
- If S is the support of the random variable, then [1]

$$\int_{x \in S} f(x)\, dx = 1$$

  and any function with this property is a pmf.

- Probabilities are obtained by integrating (visualized by the area under the density curve):

$$\Pr(a \leq X \leq b) = \int_a^b f(x)\, dx$$



kernel function: If $\int_{x \in S} f(x)\, dx = k$ for some real number k, then f is a kernel function. We can obtain the pdf from the kernel by dividing by k.

cumulative distribution function (cdf): a function (often denoted with a capital F) that takes a possible value of a random variable as input and returns the probability of obtaining a value less than or equal to the input:

$$F_X(x) = \Pr(X \leq x)$$

cdfs can be defined for both discrete and continuous random variables.

a family of distributions: is a collection of distributions which share common features but are distinguished by different parameter values. For example, we could have the family of distributions of fair dice random variables. The parameter would tell us how many sides the die has. Statisticians call this family the **discrete uniform distributions** because all the probabilities are equal (1/6 for 6-sided die, 1/10 for a D_10, etc.).

We will get to know several important families of distributions, among them the **binomial**, **beta**, **normal**, and **t** families will be especially useful. You may already be familiar with some or all of these. We will also use distributions that have no name and are only described by a pmf or pdf, or perhaps only by a large number of random samples from which we attempt to estimate the pmf or pdf.

[1] Kruschke likes to write his integrals in a different order: $\int dx\, f(x)$ instead of $\int f(x)\, dx$. Either order means the same thing.

4.2 Distributions in R

pmfs, pdfs, and cdfs are available in R for many important families of distributions. You just need to know a few things:

- each family has a standard abbreviation in R
- pmf and pdf functions begin with the letter `d` followed by the family abbreviation
- cdf functions begin with the letter `p` followed by the family abbreviation
- the inverse of the cdf function is called a quantile function, it starts with the letter `q`
- functions beginning with `r` can generate random samples from a distribution
- help for any of these functions will tell you what R calls the parameters of the family.
- `gf_dist()` can be used to make various plots of distributions.

4.2.1 Example: Normal distributions

As an example, let's look the family of normal distributions. If you type `dnorm(` and then hit TAB or if you type `args(dnorm)` you can see the arguments for this function.

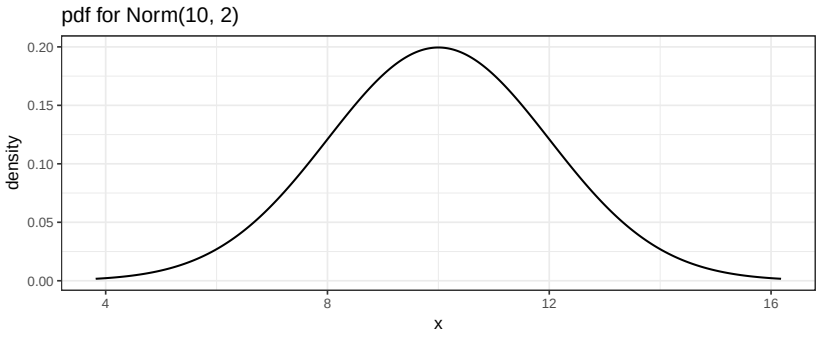
```
args(dnorm)
```

```
## function (x, mean = 0, sd = 1, log = FALSE)
## NULL
```

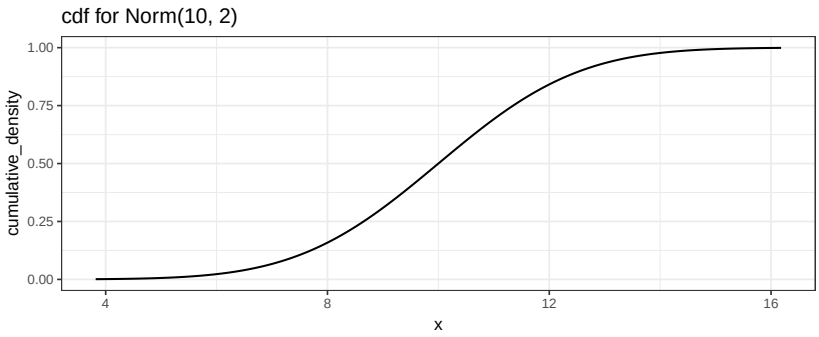
From this we see that the parameters are called `mean` and `sd` and have default value of 0 and 1. These values will be used if we don't specify something else. As with many of the pmf and pdf functions, there is also an option to get back the log of the pmf or pdf by setting `log = TRUE`. This turns out to be computationally much more efficient in many contexts, as we will see.

Let's begin with some pictures of a normal distribution with mean 10 and standard deviation 1:

```
gf_dist("norm", mean = 10, sd = 2, title = "pdf for Norm(10, 2)")
```




```
gf_dist("norm", mean = 10, sd = 2, kind = "cdf", title = "cdf for Norm(10, 2)")
```



Now some exercises. Assume $X \sim \mathsf{Norm}(10, 2)$.

1. What is $\Pr(X \leq 5)$?

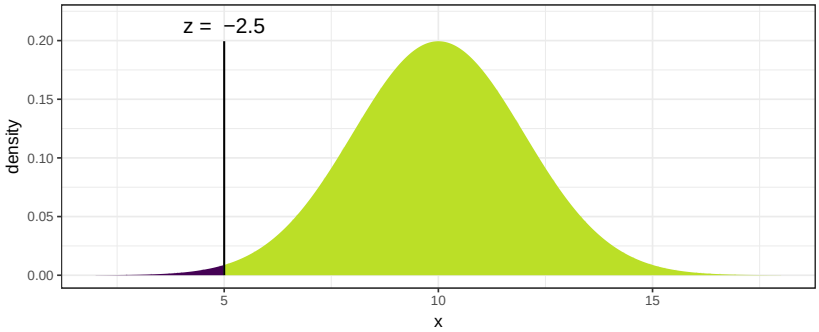
   We can see by inspection that it is less that 0.5. `pnorm()` will give us the value we are after; `xpnorm()` will provide more verbose output and a plot as well.

```
pnorm(5, mean = 10, sd = 2)
```

```
## [1] 0.00621
```

```
xpnorm(5, mean = 10, sd = 2)
```

```
##
## If X ~ N(10, 2), then
##  P(X <= 5) = P(Z <= -2.5) = 0.00621
##  P(X >  5) = P(Z >  -2.5) = 0.9938
##
```




```
## [1] 0.00621
```

2. What is $\Pr(5 \leq X \leq 10)$?

```
pnorm(10, mean = 10, sd = 2) - pnorm(5, mean = 10, sd = 2)
```

```
## [1] 0.4938
```

3. How tall is the density function at it's peak?

   Normal distributions are symmetric about their means, so we need the value of the pdf at 10.

```
dnorm(10, mean = 10, sd = 2)
```

```
## [1] 0.1995
```

4. What is the mean of a Norm(10, 2) distribution?

   Ignoring for the moment that we know the answer is 10, we can compute it. Notice the use of `dnorm()` in the computation.

```
integrate(function(x) x * dnorm(x, mean = 10, sd = 2), -Inf, Inf)
```

```
## 10 with absolute error < 0.0011
```

5. What is the variance of a Norm(10, 2) distribution?

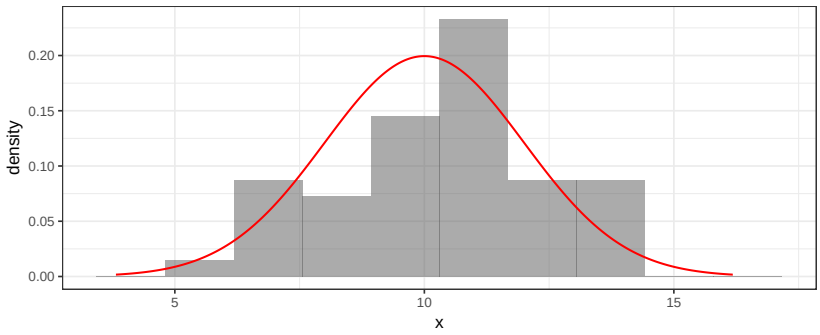
   Again, we know the answer is the square of the standard deviation, so 4. But let's get R to compute it in a way that would work for other distributions as well.

```
integrate(function(x) (x - 10)^2 * dnorm(x, mean = 10, sd = 2), -Inf, Inf)
```

```
## 4 with absolute error < 7.1e-05
```

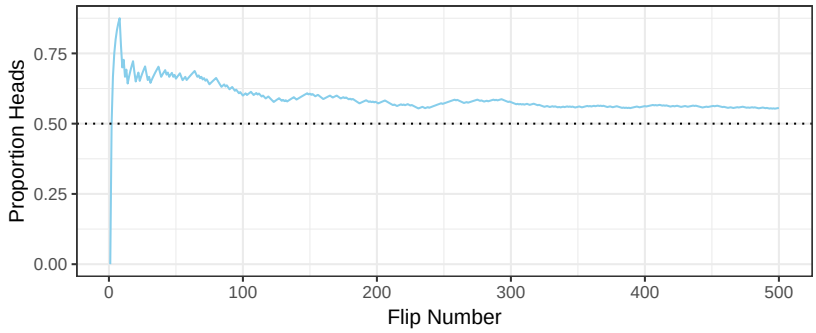
6. Simulate a data set with 50 values drawn from a Norm(10, 2) distribution and make a histogram of the results and overlay the normal pdf for comparison.

```
x <- rnorm(50, mean = 10, sd = 2)
# be sure to use a density histogram so it is on the same scale as the pdf!
gf_dhistogram(~ x, bins = 10) %>%
  gf_dist("norm", mean = 10, sd = 2, color = "red")
```



4.2.2 Simulating running proportions

```
library(ggformula)
library(dplyr)
theme_set(theme_bw())
Flips <-
  tibble(
    n = 1:500,
    flip = rbinom(500, 1, 0.5),
    running_count = cumsum(flip),
    running_prop  = running_count / n
  )

gf_line(
  running_prop ~ n, data = Flips,
  color = "skyblue",
  ylim = c(0, 1.0),
  xlab = "Flip Number", ylab = "Proportion Heads",
  main = "Running Proportion of Heads") %>%
  gf_hline(yintercept = 0.5, linetype = "dotted")
```



4.3 Joint, marginal, and conditional distributions

Sometimes (most of the time, actually) we are interested joint distributions. A joint distribution is the distribution of multiple random variables that result from the same random process. For example, we might roll a pair of dice and obtain two numbers (one for each die). Or we might collect a random sample of people and record the height for each of them. Or we might randomly select one person, but record multiple facts (height and weight, for example). All of these situations are covered by joint distributions.[2]

4.3.1 Example: Hair and eye color

Kruschke illustrates joint distributions with an example of hair and eye color recorded for a number of people. [3] That table below has the proportions for each hair/eye color combination. For example,

Hair/Eyes	Blue	Green	Hazel	Brown
Black	0.034	0.115	0.008	0.025
Blond	0.159	0.012	0.027	0.017
Brown	0.142	0.201	0.049	0.091
Red	0.029	0.044	0.024	0.024

Each value in the table indicates the proportion of people that have a particular hair color *and* a particular eye color. So the upper left cell says that 3.4% of people have black hair and blue eyes (in this particular sample – the proportions will vary a lot depending on the population of interest). We will denote this as

$$\Pr(\text{Hair} = \text{black}, \text{Eyes} = \text{blue}) = 0.034 \; .$$

or more succinctly as

$$p(\text{black}, \text{blue}) = 0.034 \; .$$

This type of probability is called a **joint probability** because it tells about the probability of **both** things happening.

Use the table above to do the following.

1. What is $p(\text{brown}, \text{green})$ and what does that number mean?
2. Add the proportion across each row and down each column. (Record them to the right and along the bottom of the table.) For example, in the first row we get

   $$0.034 + 0.115 + 0.008 + 0.025 = 0.182 \; .$$

   a. Explain why $p(\text{black}) = 0.182$ is good notation for this number.
3. Up to round-off error, the total of all the proportions should be 1. Check that this is true.
4. What proportion of people with black hair have blue eyes?

   This is called a conditional probability. We denote it as $\Pr(\text{Eyes} = \text{blue} \mid \text{Hair} = \text{black})$. or $p(\text{blue} \mid \text{black})$.
5. Compute some other conditional probabilities.

   a. $p(\text{black} \mid \text{blue})$.
   b. $p(\text{blue} \mid \text{blond})$.

[2] Kruschke calls these 2-way distributions, but there can be more than variables involved.

[3] The datasets package has a version of this data with a third variable: `sex`. (It is as a 3d table rather than as a data frame). According to the help for this data set, these data come from "a survey of students at the University of Delaware reported by Snee (1974). The split by Sex was added by Friendly (1992a) for didactic purposes." It isn't exactly clear what population this might represent

c. $p(\text{blond} \mid \text{blue})$.
d. $p(\text{brown} \mid \text{hazel})$.
e. $p(\text{hazel} \mid \text{brown})$.

6. There are 32 such conditional probabilities that we can compute from this table. Which is largest? Which is smallest?
7. Write a general formula for computing the conditional probability $p(c \mid r)$ from the $p(r, c)$ values. (r and c are to remind you of rows and columns.)
8. Write a general formula for computing the conditional probability $p(r \mid c)$ from the $p(r, c)$ values.

If we have continuous random variables, we can do a similar thing. Instead of working with probability, we will work with a pdf. Instead of sums, we will have integrals.

8. Write a general formula for computing each of the following if $p(x, y)$ is a continuous joint pdf.

a. $p_X(x) = p(x) =$
b. $p_Y(y) = p(y) =$
c. $p_{Y|X}(y \mid x) = p(y \mid x) =$
d. $p_{X|Y}(y \mid x) = (x \mid y) =$

9. We can expression both versions of conditional probability using a word equation. Fill in the missing numerator and denominator

$$\text{conditional} = \frac{\qquad}{\qquad}$$

4.3.2 Independence

If $p(x \mid y) = p(x)$ (conditional = marginal) for all combinations of x and y, we say that X and Y are **independent**.

10. Use the definitions above to express independence another way.
11. Are hair and eye color independent in our example?
12. True or False. If we randomly select a card from a standard deck (52 cards, 13 denominations, 4 suits), are suit and denomination independent?
13. Create a table for two independent random variables X and Y, each of which takes on only 3 possible values.
14. Now create a table for a different pair X and Y that are not independent but have the same marginal probabilities as in the previous exercise.

4.4 Exercises

1. Suppose a random variable has the pdf $p(x) = 6x(1 - x)$ on the interval $[0, 1]$. (That means it is 0 outside of that interval.)
   a. Use `function()` to create a function in R that is equivalent to `p(x)`.
   b. Use `gf_function()` to plot the function on the interval $[0, 1]$.
   c. Integrate by hand to show that the total area under the pdf is 1 (as it should be for any pdf).
   d. Now have R compute that same integral (using `integrate()`).
   e. What is the largest value of $p(x)$? At what value of x does it occur? Is it a problem that this value is larger than 1?
   Hint: differentiation might be useful.

2. Recall that $\mathrm{E}(X) = \int x f(x)\, dx$ for a continuous random variable with pdf f and $\mathrm{E}(X) = \sum x f(x)\, dx$ for a discrete random variable with pmf f. (The integral or sum is over the support of the random variable.) Compute the expected value for the following random variables.
   a. A is discrete with pmf $f(x) = x/10$ for $x \in \{1, 2, 3, 4\}$.
   b. B is continuous with kernel $f(x) = x^2(1 - x)$ on $[0, 1]$.
   Hint: first figure out what the pdf is.
3. Compute the variance and standard deviation of each of the distributions in the previous problem.
4. In Bayesian inference, we will often need to come up with a distribution that matches certain features that correspond to our knowledge or intuition about a situation. Find a normal distribution with a mean of 10 such that half of the distribution is within 3 of 10 (ie, between 7 and 13).

   Hint: use `qnorm()` to determine how many standard deviations are between 10 and 7.
5. School children were surveyed regarding their favorite foods. Of the total sample, 20% were 1st graders, 20% were 6th graders, and 60% were 11th graders. For each grade, the following table shows the proportion of respondents that chose each of three foods as their favorite.
   a. From that information, construct a table of joint probabilities of grade and favorite food.
   b. Are grade and favorite food independent? Explain how you ascertained the answer.

grade	Ice cream	Fruit	French fries
1st	0.3	0.6	0.1
6th	0.6	0.3	0.1
11th	0.3	0.1	0.6

6. Three cards are placed in a hat. One is black on both sides, one is white on both sides, and the third is white on one side and black on the other. One card is selected at random from the three cards in the hat and placed on the table.
   The top of the card is black.
   a. What is the probability that the bottom is also black?
   b. What notation should we use for this probability?
7. The three cards from the previous problem are returned to the hat. Once again a card is pulled and placed on the table, and once again the top is black. This time a second card is drawn and placed on the table. The top side of the second card is white.
   a. What is the probability that the bottom side of the first card is black?
   b. What is the probability that the bottom side of the second card is black?
   c. What is the probability that the bottom side of both cards is black?
8. **Pandas.** Suppose there are two species of panda, A and B. Without a special blood test, it is not possible to tell them apart. But it is known that half of pandas are of each species and that 10% of births from species A are twins and 20% of births from species B are twins.
   a. If a female panda has twins, what is the probability that she is from species A?
   b. If the same panda later has another set of twins, what is the probability that she is from species A?
   c. A different panda has twins and a year later gives birth to a single panda. What is the probability that this panda is from species A?
9. **More Pandas.** You get more interested in pandas and learn that at your favorite zoo, 70% of pandas are species A and 30% are species B. You learn that one of the pandas has twins.
   a. What is the probability that the panda is species A?
   b. The same panda has a single panda the next year. Now what is the probability that the species is A?

4.5 Footnotes

Chapter 5

Bayes' Rule and the Grid Method

5.1 The Big Bayesian Idea

- Model specifies

$$\begin{aligned} \text{prior:} \quad & p(\text{parameter values}) \\ \text{likelihood:} \quad & p(\text{data values} \mid \text{parameter values}) \end{aligned}$$

- Bayes rule + data gives

$$\text{posterior:} \quad p(\text{parameter values} \mid \text{data values})$$

Let's let D be the data values and θ the parameter values. So the prior is $p(\theta)$, and the posterior is $p(\theta \mid D)$. Recall that

$$\begin{aligned} p(D, \theta) &= p(\theta) \cdot p(D \mid \theta) \\ &= p(D) \cdot p(\theta \mid D) \end{aligned}$$

$$p(D) \cdot p(\theta \mid D) = p(\theta) \cdot p(D \mid \theta)$$

Solving that last equation for $p(\theta \mid D)$ gives

$$\begin{aligned} p(\theta \mid D) &= \frac{p(\theta) \cdot p(D \mid \theta)}{p(D)} \\ &= \frac{p(\theta) \cdot p(D \mid \theta)}{\sum_{\theta^*} p(\theta^*) p(D \mid \theta^*)} \text{or} \frac{p(\theta) \cdot p(D \mid \theta)}{\int p(\theta^*) p(D \mid \theta^*) \, d\theta^*} \end{aligned}$$

Important facts about the denominator:

- The denominator sums or integrates over all possible numerators.
- The denominator depends on D but not on θ.

- So it is just a normalizing constant that guarantees that total probability is 1 for the posterior distribution. That is, it converts the kernel into a pdf.
- If we only need a kernel, we don't need to compute the denominator.

Another way of saying all this is that

$$p(\theta \mid D) \propto p(\theta) \cdot p(D \mid \theta)$$

$$\text{posterior} \propto \text{prior} \cdot \text{likelihood}$$

$$\text{kernel of posterior} = \text{prior} \cdot \text{likelihood}$$

That last line is worth repeating. It's the most important equation in this course:

$$\text{(kernel of) posterior} = \text{prior} \cdot \text{likelihood}$$

5.1.1 Likelihood

For a fixed data set D, $p(\theta)$ and $p(\theta \mid D)$ are pdfs (or pmfs) describing the prior and posterior distributions of θ.

The likelihood function is different. If we consider $p(D \mid \theta)$ to be a function of D, it is not a pdf or pmf, and the total area under the curve for all possible data sets need not be 1.

The likelihood function is specified by the model. The model must tell us "how likely a given data set would be" for a specified value of the parameters θ.

5.1.2 When Bayes is easy

If the number of possible values for θ is small (so we could just do all the arithmetic by brute force) or if the integrals and sums are easy to compute, then Bayesian updating (computing the posterior) is relatively easy. We'll start with examples (at least approximately) in those two happy situations and worry about some of the complications a little bit later.

5.2 Estimating the bias in a coin using the Grid Method

Big ideas:

1. Discretize the parameter space if it is not already discrete.
2. Compute prior and likelihood at each "grid point" in the (discretized) parameter space.
3. Compute (kernel of) posterior as prior · likelihood at each "grid point".
4. Normalize to get posterior, if desired.

Below we will see how to perform these four steps in R.

5.2.1 Creating a Grid

The parameter is θ and we will discretize by selecting 1001 grid points from 0 to 1 by 0.001.[1]

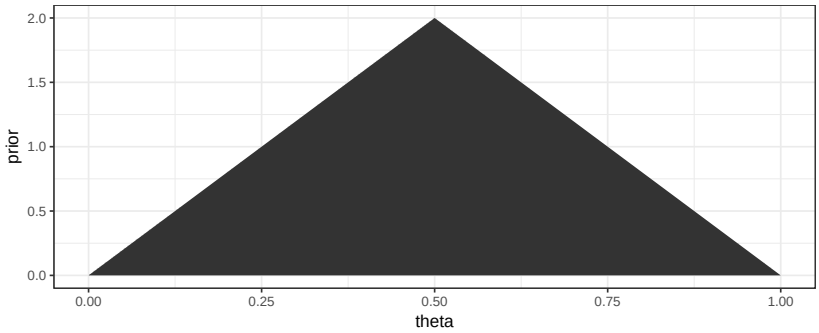
[1] There are other ways to do this, but `expand.grid()` will work when we have more than one parameter, so we'll start using it already in the simpler case of a single parameter.

```
CoinsGrid <-
  expand.grid(
    theta = seq(0, 1, by = 0.001)
  )
head(CoinsGrid)
```

theta
0.000
0.001
0.002
0.003
0.004
0.005

Now let's add on a triangle prior with a peak when $\theta = 0.5$.

```
library(triangle)
CoinsGrid <-
  expand.grid(
    theta = seq(0, 1, by = 0.001)
  ) %>%
  mutate(
    prior = dtriangle(theta)        # triangle distribution
  )
gf_area(prior ~ theta, data = CoinsGrid)
```



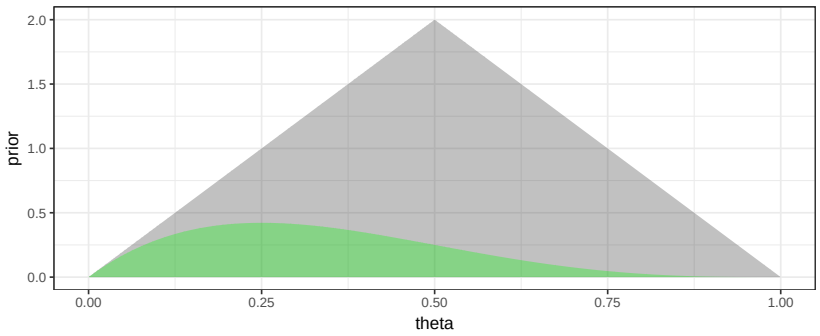
Now the likelihood for a small data set: 1 success out of 4 trials. This is the trickiest part. `dbinom(x, size = n, prob = theta)` will calculate the probability that we want for a given value of `x`, `n`, and `theta`. We want to do this

- for each value of `theta`
- but using the same values for `x` and `n` each time

`purrr:map_dbl()` helps us tell R how to do this. Each value of `theta` gets plugged in for `.x` and a vector of numbers (dbl stands for double – computer talk for real number) is returned.

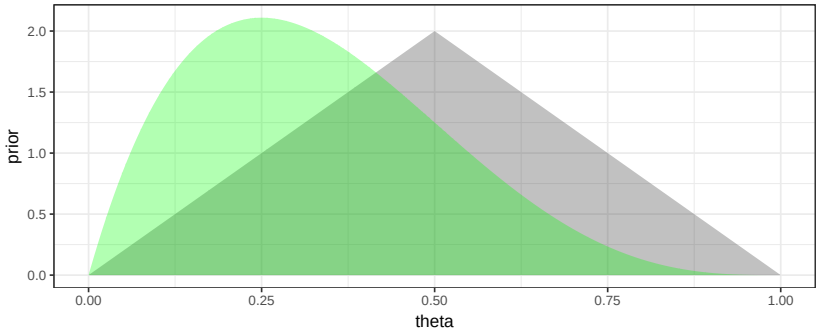
```
library(purrr)
x <- 1; n <- 4
CoinsGrid <-
  expand.grid(
    theta = seq(0, 1, by = 0.001)
  ) %>%
  mutate(
    prior = dtriangle(theta),        # triangle distribution
    likelihood = map_dbl(theta, ~ dbinom(x = x, size = n, .x))
 )
```

```
gf_area( prior ~ theta, data = CoinsGrid, alpha = 0.3) %>%
  gf_area( likelihood ~ theta, data = CoinsGrid, alpha = 0.3, fill = "green")
```



Note: Likelihoods are NOT pmfs or pdfs, so the total area under a likelihood function is usually not 1. We can make a normalized version for the purpose of plotting. (Recall, we will normalize the posterior at the end anyway, so it is fine if the likelihood is off by a constant multiple at this point in the process.) We do this by dividing by sum of the likelihoods and by the width of the spaces between grid points.

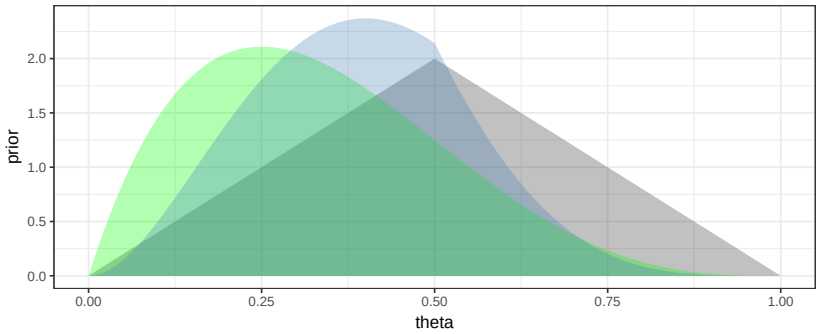
```
library(purrr)
x <- 1; n <- 4
CoinsGrid <-
  expand.grid(
    theta = seq(0, 1, by = 0.001)
  ) %>%
  mutate(
    prior = dtriangle(theta),         # triangle distribution
    likelihood  = map_dbl(theta, ~ dbinom(x = x, size = n, .x)),
    likelihood1 = likelihood / sum(likelihood) / 0.001    # "normalized"
 )
gf_area( prior ~ theta, data = CoinsGrid, alpha = 0.3) %>%
  gf_area( likelihood1 ~ theta, data = CoinsGrid, alpha = 0.3, fill = "green")
```



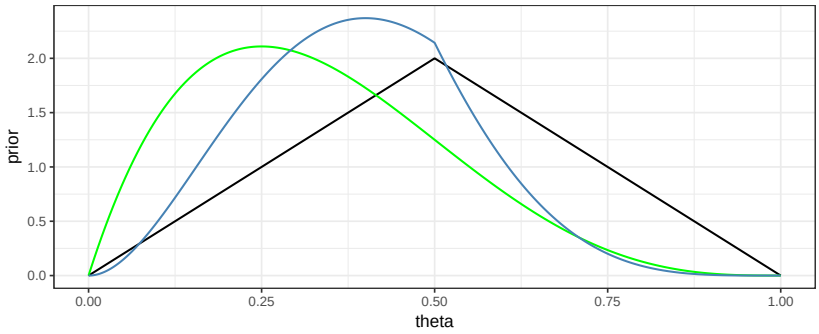
The hardest part of the coding (computing the likelihood) is now done. Getting the posterior is as simple as computing a product.

```
library(purrr)
x <- 1; n <- 4
CoinsGrid <-
  expand.grid(
    theta = seq(0, 1, by = 0.001)
  ) %>%
  mutate(
    prior = dtriangle(theta),         # triangle distribution
    likelihood  = map_dbl(theta, ~ dbinom(x = x, size = n, .x)),
```

```
    likelihood1 = likelihood / sum(likelihood) / 0.001, # "normalized"
    posterior0 = prior * likelihood,                    # unnormalized
    posterior = posterior0 / sum(posterior0) / 0.001    # normalized
  )

gf_area( prior ~ theta, data = CoinsGrid, alpha = 0.3) %>%
  gf_area( likelihood1 ~ theta, data = CoinsGrid, alpha = 0.3, fill = "green") %>%
  gf_area( posterior ~ theta, data = CoinsGrid, alpha = 0.3, fill = "steelblue")
```




```
gf_line( prior ~ theta, data = CoinsGrid) %>%
  gf_line( likelihood1 ~ theta, data = CoinsGrid, color = "green") %>%
  gf_line( posterior ~ theta, data = CoinsGrid, color = "steelblue")
```



5.2.2 HDI from the grid

The **CalvinBayes** packages includes a function `hdi_from_grid()` to compute highest density intervals from a grid. The basic idea of the algorithm used is to sort the grid by the posterior values. The mode will be at the end of the list, and the "bottom 95%" will be the HDI (or some other percent if we choose a different `level`). This method works as long as the posterior is unimodal, increasing to the mode from either side.

`hdi_from_grid()` is slightly more complicated because it handles things like multiple parameters and performs some standardization (so we can work with kernels, for example). It does assume that the grid is uniform (ie, evenly spaced).

We simply provide

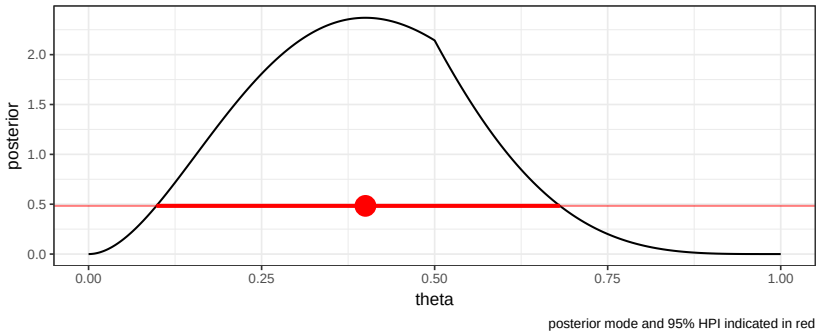
- the data frame containing our grid calculations,
- `pars`: the name of the parameter (or parameters) for which we want intervals (default is the first column in the grid),
- `prob`: the probability we want in covered by our interval (0.95 by default),
- `posterior`: the name of the column containing the posterior kernel values (`"posterior"` by default)

```
library(CalvinBayes)
hdi_from_grid(CoinsGrid, pars = "theta", prob = 0.95)
```

param	lo	hi	prob	height	mode_height	mode
theta	0.098	0.681	0.9501	0.4833	2.37	0.4

With this information in hand, we can add a representation of the 95% HDI to our plot.

```
HDICoins <- hdi_from_grid(CoinsGrid, pars = "theta", prob = 0.95)
gf_line(posterior ~ theta, data = CoinsGrid) %>%
  gf_hline(yintercept = ~height, data = HDICoins,
           color = "red", alpha = 0.5) %>%
  gf_pointrangeh(height ~ mode + lo + hi, data = HDICoins,
             color = "red", size = 1) %>%
  gf_labs(caption = "posterior mode and 95% HPI indicated in red")
```



5.2.3 Automating the grid

Note: This function is a bit different from `CalvinBayes::BernGrid()`.

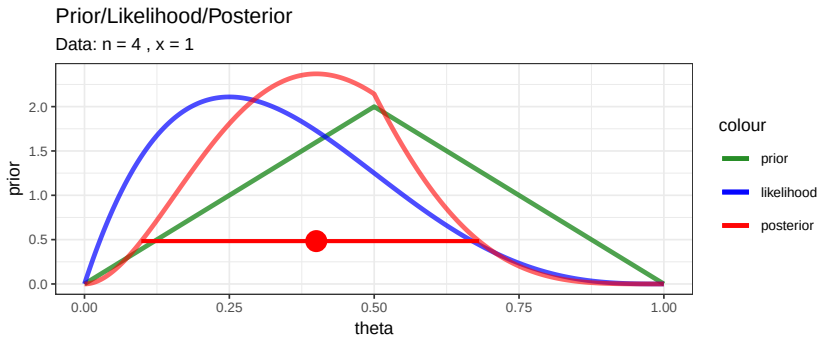
```
MyBernGrid <- function(
  x, n,                    # x successes in n tries
  prior = dunif,
  resolution = 1000,       # number of intervals to use for grid
  ...) {

  Grid <-
    expand.grid(
      theta = seq(0, 1, length.out = resolution + 1)
      ) %>%
    mutate( # saving only the normalized version of each
      prior = prior(theta, ...),
      prior = prior / sum(prior) * resolution,
      likelihood  = dbinom(x, n, theta),
      likelihood = likelihood / sum(likelihood) * resolution,
      posterior = prior * likelihood,
      posterior  = posterior / sum(posterior) * resolution
    )

  H <- hdi_from_grid(Grid, pars = "theta", prob = 0.95)

  gf_line(prior ~ theta, data = Grid, color = ~"prior",
          size = 1.15, alpha = 0.8) %>%
  gf_line(likelihood ~ theta, data = Grid, color = ~"likelihood",
```

```
          size = 1.15, alpha = 0.7) %>%
  gf_line(posterior ~ theta, data = Grid, color = ~"posterior",
          size = 1.15, alpha = 0.6) %>%
    gf_pointrangeh(
      height ~ mode + lo + hi, data = H,
      color = "red", size = 1) %>%
    gf_labs(title = "Prior/Likelihood/Posterior",
            subtitle = paste("Data: n =", n, ", x =", x)) %>%
    gf_refine(
      scale_color_manual(
        values = c(
          "prior" = "forestgreen",
          "likelihood" = "blue",
          "posterior" = "red"),
        breaks = c("prior", "likelihood", "posterior")
        )) %>%
    print()
  invisible(Grid)    # return the Grid, but don't show it
}
```

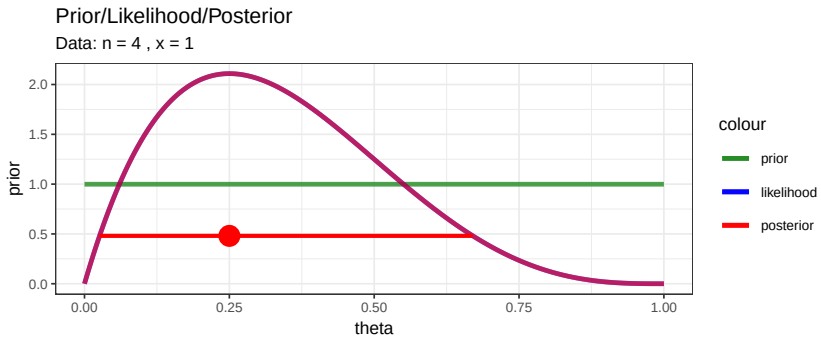
This function let's us quickly explore several scenarios and compare the results.

- How does changing the prior affect the posterior?
- How does changing the data affect the posterior?

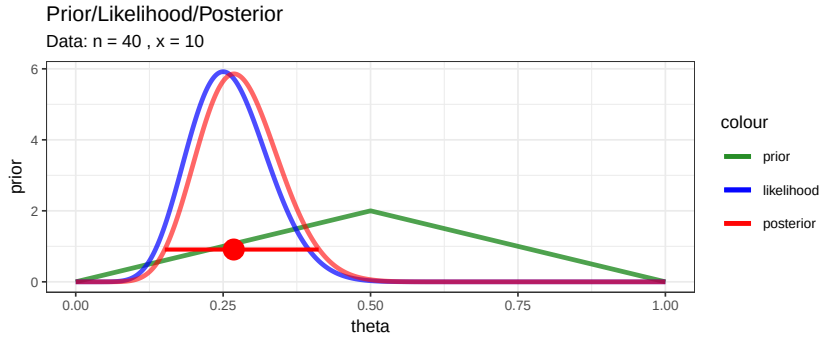
```
library(triangle)
MyBernGrid(1, 4, prior = dtriangle, a = 0, b = 1, c = 0.5)
```



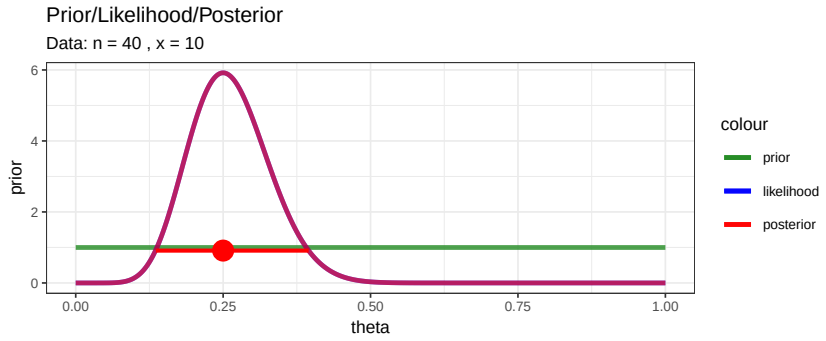

```
MyBernGrid(1, 4, prior = dunif)
```



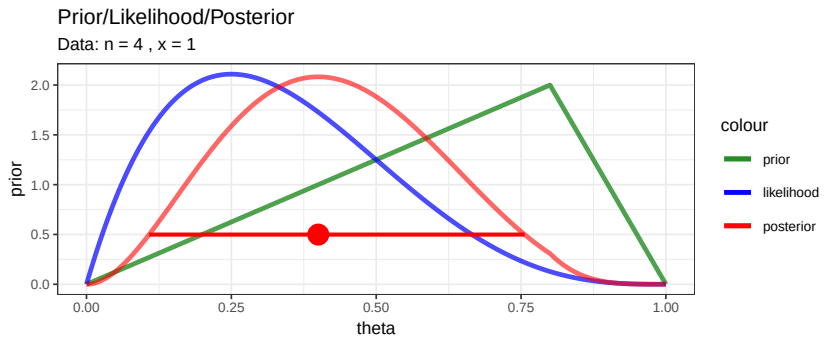
```
MyBernGrid(10, 40, prior = dtriangle, a = 0, b = 1, c = 0.5)
```



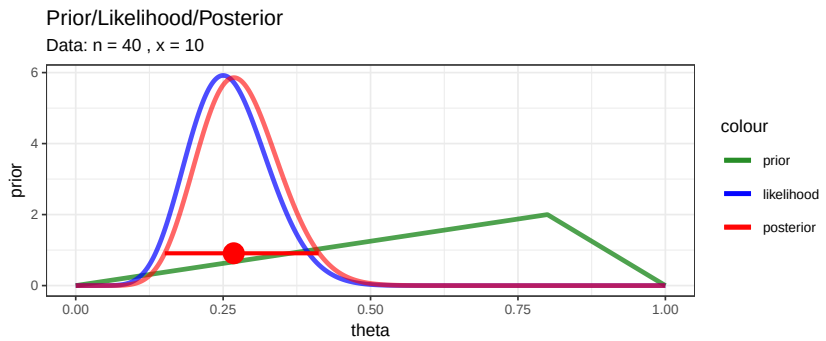

```
MyBernGrid(10, 40, prior = dunif)
```



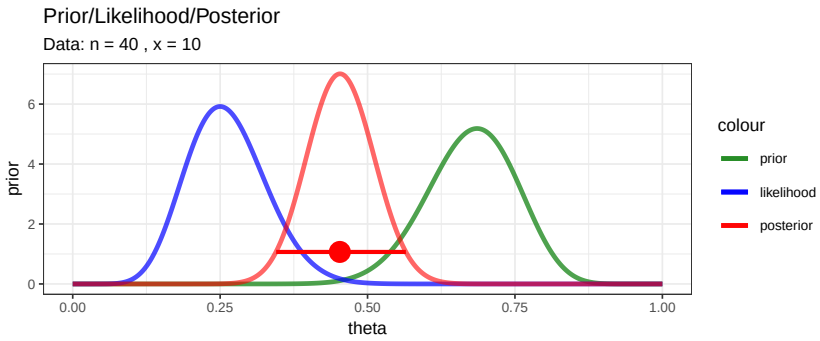

```
MyBernGrid(1, 4, prior = dtriangle, a = 0, b = 1, c = 0.8)
```




```
MyBernGrid(10, 40, prior = dtriangle, a = 0, b = 1, c = 0.8)
```




```
MyBernGrid(10, 40, prior = dbeta, shape1 = 25, shape2 = 12)
```



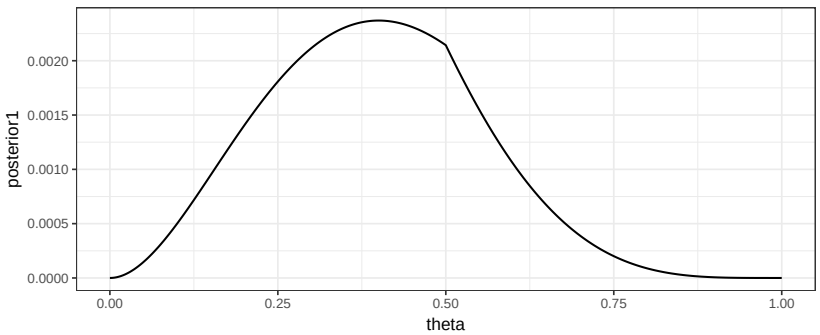
5.3 Working on the log scale

Very often it is numerically better to work on the log scale, computing the logs of the prior, likelihood, and posterior. There are at least two reasons for this:

- Likelihoods are often very small, especially if there is a lot of data. (Even with the most credible parameter values, the probability of getting exactly the data set that was observed is very low.)
- Likelihoods often involve products and exponentiation. Take logarithms turns these into sums and products. In fact, we can often compute these logs without first computing the prior, likelihood, or posterior. (It depends on the form of those functions.)

Let's redo our previous example working on the log scale until the very end.

```
x <- 1; n <- 4
CoinsGridLog <-
  expand.grid(
    theta = seq(0, 1, by = 0.001)
  ) %>%
  mutate(
    logprior = log(dtriangle(theta, 0, 1)),       # triangle distribution
    loglik  =
      map_dbl(theta, ~ dbinom(x = x, size = n, prob = .x, log = TRUE)),
    logpost = logprior + loglik,
    posterior = exp(logpost),
    posterior1 = posterior / sum(posterior, na.rm = TRUE)
  )
gf_line(posterior1 ~ theta, data = CoinsGridLog)
```



5.4 Discrete Parameters

Most of the parameters we will encounter will be able to take on all values in some interval. If we have parameter that can only take on a discrete set of values, then the grid method is exact if we make a grid point for each of those values. Usually we think of parameters taking on numerical values, but here is an example where a parameter takes on categorical labels for values.

Suppose we have a medical test for a disease and we know the following information.

- Only 1 person in 1000 has the disease in our population of interest
- If a person is healthy, the probability that the test will be correct is 97%. (97% = **specificity** = true negative rate)
- If a person is diseased, the probability that the test will be correct is 99%. (99% = **sensitivity** = true positive rate)

If a person is tested and the test comes back "positive" (indicating disease), what is the probability that the person actually has the disease?

Our parameter that we want to estimate is disease status, which can take on only two possible values: healthy or diseased.

```
Disease_Grid <-
  tibble(
    status = c("healthy", "sick"),
    prior = c(999/1000, 1/1000)
  )
Disease_Grid
```

status	prior
healthy	0.999
sick	0.001

Now let's add in the likelihood and posterior information assuming a positive test result.

```
Disease_Grid <-
  tibble(
    status = c("healthy", "sick"),
    prior = c(999/1000, 1/1000),
    likelihood = c(.03, .99),
    posterior = prior * likelihood,
    posterior1 = posterior / sum(posterior)
  )
Disease_Grid
```

status	prior	likelihood	posterior	posterior1
healthy	0.999	0.03	0.030	0.968
sick	0.001	0.99	0.001	0.032

So we have updated our belief from a 0.1% chance the person has the disease to a 3.2$ chance that they have the disease. That's a sizable increase, but the person is still most likely healthy, not diseased.

5.5 Exercises

1. More testing.
    a. Suppose that the population consists of 100,000 people. Compute how many people would be expected to fall into each cell of Table 5.4 on page 104 of *DBDA2e*. (To compute the expected number of people in a cell, just multiply the cell probability by the size of the population.)

You should find that out of 100,000 people, only 100 have the disease, while 99,900 do not have the disease. These marginal frequencies instantiate the prior probability that $p(\theta = \frown) = 0.001$. Notice also the cell frequencies in the column $\theta = \frown$, which indicate that of 100 people with the disease, 99 have a positive test result and 1 has a negative test result. These cell frequencies instantiate the hit rate of 0.99. Your job for this part of the exercise is to fill in the frequencies of the remaining cells of the table.

b. Take a good look at the frequencies in the table you just computed for the previous part. These are the so-called "natural frequencies" of the events, as opposed to the somewhat unintuitive expression in terms of conditional probabilities (Gigerenzer & Hoffrage, 1995). From the cell frequencies alone, determine the proportion of people who have the disease, given that their test result is positive.

Your answer should match the result from applying Bayes' rule to the probabilities.

c. Now we'll consider a related representation of the probabilities in terms of natural frequencies, which is especially useful when we accumulate more data. This type of representation is called a "Markov" representation by Krauss, Martignon, and Hoffrage (1999). Suppose now we start with a population of $N = 10,000,000$ people. We expect 99.9% of them (i.e., 9,990,000) not to have the disease, and just 0.1% (i.e., 10,000) to have the disease. Now consider how many people we expect to test positive. Of the 10,000 people who have the disease, 99%, (i.e., 9,900) will be expected to test positive. Of the 9,990,000 people who do not have the disease, 5% (i.e., 499,500) will be expected to test positive. Now consider re-testing everyone who has tested positive on the first test. How many of them are expected to show a negative result on the re-test?

d. What proportion of people who test positive at first and then negative on retest, actually have the disease? In other words, of the total number of people at the bottom of the diagram in the previous part (those are the people who tested positive then negative), what proportion of them are in the left branch of the tree? How does the result compare with your answer to Exercise 5.1?

2. Suppose we have a test with a 97% specificity and a 99% sensitivity just like in Section 5.4. Now suppose that a random person is selected, has a first test that is positive, then is retested and has a second test that is negative.
Taking into account both tests, and assuming the results of the two tests are independent, what is the probability that the person has the disease?

   Hint: We can use the the posterior after the first test as a prior for the second test. Be sure to keep as many decimal digits as possible (use R and don't round intermediate results).

   Note: In this problem we are assuming the the results of the two tests are independent, which might not be the case for some medical tests.

3. Consider again the disease and diagnostic test of the previous exercise and Section 5.4.

   a. Suppose that a person selected at random from the population gets the test and it comes back negative. Compute the probability that the person has the disease.

   b. The person then gets re-tested, and on the second test the result is positive. Compute the probability that the person has the disease.

   c. How does the result compare with your answer in the previous exercise?

4. Modify `MyBernGrid()` so that it takes an argument specifying the probability for the HDI. Use it to create a plot showing 50% HDI for theta using a symmetric triangle prior and data consisting of 3 success and 5 failures.

5. Let's try the grid method for a model with two parameters. Suppose we want to estimate the mean and standard deviation of the heights of 21-year-old American men or women (your choice which group). First, lets get some data.

```
library(NHANES)
Men   <- NHANES %>% filter(Gender == "male", Age == 21)
Women <- NHANES %>% filter(Gender == "female", Age == 21)
```

Likelihood Our model is that heights are normally distributed with mean μ and standard deviation σ:

Prior. For our prior, let's use something informed just a little bit by what we know about people (we could do better with other priors, but that's not our goal at the moment):

- the mean height is somewhere between 5 and 7 feet (let's use 150 and 200 cm which are close to that)
- the standard deviation is positive, but no more than 20 cm (so 95% of people are within 40 cm (~ 16 inches) of average – that seems like a pretty safe bet).
- we will use a uniform prior over these ranges (even though you probably believe that some parts of the ranges are much more credible than others).

So our model is

$$\begin{aligned} \text{Height} &\sim \mathsf{Norm}(\mu, \sigma) \\ \mu &\sim \mathsf{Unif}(150, 200) \\ \sigma &\sim \mathsf{Unif}(0, 20) \end{aligned}$$

Grid. Use a grid that has 200-500 values for each parameter. Fill in the ?? below to create and update your grid.

Notes:

- It is more numerically stable to work on the log-scale as much as possible,
- You may normalize if you want to, but it isn't necessary for this problem.

```
library(purrr)
Height_Grid <-
  expand.grid(
    mu      = seq(??, ??, ?? = ??),
    sigma   = seq(??, ??, ?? = ??)
  ) %>%
  filter(sigma != 0) %>%   # remove sigma = 0
  mutate(
    prior = ??,
    logprior = ??,
    loglik =
      map2_dbl(mu, sigma,
               ~ ?? )      # use .x for mu and .y for sigma
    logpost = logprior + loglik,
    posterior = exp(logpost)
  )
```

Once you have created and updated your grid, you can visualize your posterior using a command like this (use `gf_lims()` if you want to zoom in a bit):

```
gf_tile(posterior ~ mu + sigma, data = Height_Grid) %>%
  gf_contour(posterior ~ mu+ sigma, data = Height_Grid, color = "yellow")
```

Now answer the following questions.

a. Using the picture you just created, what would you say are credible values for μ and for σ?

b. Now use `hdi_from_grid()` to compute a 90% highest density (posterior) intervals for each parameter. Do those make sense when you compare to the picture?

c. Create a plot showing the posterior distribution for μ and a 90% HDI for μ. [Hint: how can you get a grid for the marginal distribution of μ from the grid for μ and σ?]

6. Redo the previous problem using a triangle prior for each parameter. You may choose where to put the peak of the triangle.

7. Bob plays basketball.
He shoots 70% of his shots from 2-point range and makes 48% of these shots. He shoots 30% of his shots from 3-point range and makes 32% of these shots. Joe just made a shot. What is the probability that it was a 3-point shot?

   Do this problem twice. The first time, use probability rules, carefully denoting the probabilities involved. (For any number that appears, I should be able to tell from your notation where it came from.) The second time, use the "grid method".

8. Alice has 3 hats labeled with the letters H, A, and T. In each hat are marbles of various colors.

Hat	White marbles	Red marbles	Yellow marbles
H	4	10	6
A	6	12	2
T	5	3	2

   Alice randomly selects a hat by flipping two coins. If both are heads, she chooses hat H. If both are tails, she chooses hat T. If there is one head and one tail, she chooses hat A. Once that hat is selected, she draws out two marbles.

   a. If the two marbles are both white, what is the probability that the hat was hat A?

   b. If there is one red marble and one yellow marble, what is the probability that the hat was hat A?

   c. If the two marbles are the same color, what is the probability that the hat was hat A?

5.6 Footnotes

Part II

Inferring a Binomial Probability

Chapter 6

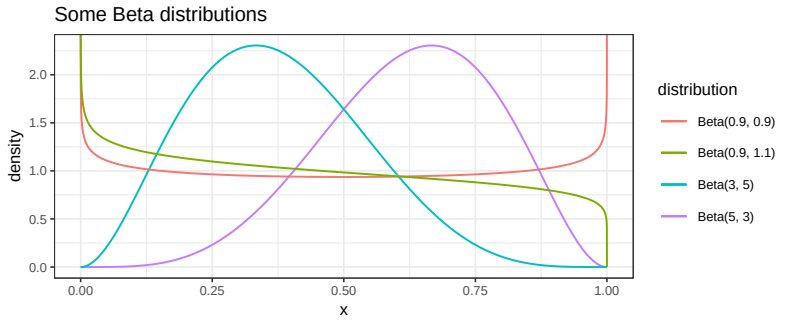
Inferring a Binomial Probability via Exact Mathematical Analysis

6.1 Beta distributions

A few important facts about beta distributions

- two parameters: $\alpha = a =$ `shape1`; $\beta = b =$ `shape2`.
- kernel: $x^{\alpha-1}(1-x)^{\beta-1}$ on $[0,1]$
- area under the kernel: $B(a,b)$ [$B()$ is the beta function, `beta()` in R]
- scaling contant: $1/B(a,b)$
- We can use `gf_dist()` to see what a beta distribution looks like.

```
gf_dist("beta", shape1 = 5, shape2 = 3, color = ~ "Beta(5, 3)") %>%
  gf_dist("beta", shape1 = 3, shape2 = 5, color = ~ "Beta(3, 5)") %>%
  gf_dist("beta", shape1 = 0.9, shape2 = 0.9, color = ~ "Beta(0.9, 0.9)") %>%
  gf_dist("beta", shape1 = 0.9, shape2 = 1.1, color = ~ "Beta(0.9, 1.1)") %>%
  gf_labs(title = "Some Beta distributions", color = "distribution")
```



6.2 Beta and Bayes

Suppose we want to estimate a proportion θ by repeating some random process (like a coin toss) N times. We will code each result using a 0 (failure) or a 1 (success): $Y_1, Y_2, \ldots, Y_N$. Here's our model.

The prior, to be determined shortly, is indicated as ??? for the moment.

$$\begin{aligned} Y_i &\sim \mathsf{Bern}(\theta) \\ \theta &\sim ??? \end{aligned}$$

6.2.1 The Bernoulli likelihood function

The first line turns into the following likelihood function – the probability of observing y_i for a give parameter value θ:

$$\begin{aligned} PrPr(Y_i = y_i \mid \theta) = p(y_i \mid \theta) &= \begin{cases} \theta & y_i = 1 \\ (1-\theta) & y_i = 0 \end{cases} \\ &= \theta^{y_i}(1-\theta)^{y_i} \end{aligned}$$

The likelihood for the entire data set is then

$$\begin{aligned} p(\langle y_1, y_2, \ldots, y_N \rangle \mid \theta) &= \prod_{i=1}^{N} \theta^{y_i}(1-\theta)^{y_i} \\ &= \theta^x(1-\theta)^{N-x} \end{aligned}$$

where x is the number of "successes" and N is the number of trials. Since the likelihood only depends on x and N, not the particular order in which the 0's and 1's are observed, we will write the likelihood as

$$p(x, N \mid \theta) = \theta^x(1-\theta)^{N-x}$$

Reminder: If we think of this expression as a function of θ for fixed data (rather than as a function of the data for fixed θ), we see that it is the kernel of a $\mathsf{Beta}(x+1, N-x+1)$ distribution. But even thought of this way, the likelihood need not be a PDF – the total sum or integral need not be 1. But we will sometimes normalize likelihood functions if we want to display them on plots with priors and posteriors.

6.2.2 A convenient prior

Now let think about our posterior:

$$\begin{aligned} p(\theta \mid x, N) &= \overbrace{p(x, N \mid \theta)}^{\text{likelihood}} \cdot \overbrace{p(\theta)}^{\text{prior}} / p(x, N) \\ &= \theta^x(1-\theta)^{N-x} \cdot p(\theta)/p(x, N) \end{aligned}$$

If we let $p(\theta) = \theta^a(1-\theta^b)$, the product is epsecially easy to evaluate:

$$\begin{aligned} p(\theta \mid x, N) &= \overbrace{p(x, N \mid \theta)}^{\text{likelihood}} \cdot \overbrace{p(\theta)}^{\text{prior}} / p(x, N) \\ &= \theta^x(1-\theta)^{N-x} \cdot \theta^a(1-\theta)^b)/p(x, N) \\ &= \theta^{x+a}(1-\theta)^{N-x+b}/p(x, N) \end{aligned}$$

In this happy situation, when mutlipying the likelihood and the prior leads to a posterior with same form as the prior, we say that the prior is a **conjugate prior** (for that particular likelihood function). So beta priors are conjugate priors for the Bernoulli likelihood, and if we use a beta prior, we will get a beta posterior and it is easy to calculate which one:

prior	data	posterior
$\mathsf{Beta(a, b)}$	x, N	$\mathsf{Beta}(x + a, N - x + b)$

6.2.3 Pros and Cons of conjugate priors

Pros: Easy and fast calculation; can reason about the relationship between prior, likelihood, and posterior based on a known distributions.

Cons: We are restricted to using a conjugate prior, and that isn't always the prior we want; many situations don't have natural conjugate priors available; the computations are often not as simple as in our current example.

6.3 Getting to know the Beta distributions

6.3.1 Important facts

You can often look up this sort of information on the Wikipedia page for a family of distributions. If you go to https://en.wikipedia.org/wiki/Beta_distribution you will find, among other things, the following:

Notation	$\text{Beta}(\alpha, \beta)$
Parameters	$\alpha > 0$ shape (real) $\beta > 0$ shape (real)
Support	$x \in [0, 1]$ or $x \in (0, 1)$
PDF	$\dfrac{x^{\alpha-1}(1-x)^{\beta-1}}{\text{B}(\alpha, \beta)}$
Mean	$\dfrac{\alpha}{\alpha + \beta}$
Mode	$\dfrac{\alpha - 1}{\alpha + \beta - 2}$ for $\alpha, \beta > 1$ 0 for $\alpha = 1, \beta > 1$ 1 for $\alpha > 1, \beta = 1$
Variance	$\dfrac{\alpha\beta}{(\alpha + \beta)^2(\alpha + \beta + 1)}$
Concentration	$\kappa = \alpha + \beta$

6.3.2 Alternative parameterizations of Beta distributions

There are several different parameterizations of the beta distributions that can be helpful in selecting a prior or interpreting a posterior.

6.3.2.1 Mode and concentration

Let the concentration be defined as $\kappa = \alpha + \beta$. Since the mode (ω) is $\dfrac{\alpha - 1}{\alpha + \beta - 2}$ for $\alpha, \beta > 1$, we can solve for α and β to get

$$\alpha = \omega(\kappa - 2) + 1 \tag{6.1}$$
$$\beta = (1 - \omega)(\kappa - 2) + 1 \tag{6.2}$$

6.3.2.2 Mean and concentration

The beta distribution may also be reparameterized in terms of its mean μ and the concentration κ. If we solve for α and β, we get

$$\alpha = \mu\kappa \tag{6.3}$$
$$\beta = (1 - \mu)\kappa \tag{6.4}$$

6.3.2.3 Mean and variance (or standard deviation)

We can also parameterize with the mean μ and variance σ^2. Solving the system of equations for mean and variance given in the table above, we get

$$\kappa = \alpha + \beta = \frac{\mu(1-\mu)}{\sigma^2} - 1 \tag{6.5}$$
$$\alpha = \mu\kappa = \mu\left(\frac{\mu(1-\mu)}{\sigma^2} - 1\right) \tag{6.6}$$
$$\beta = (1 - \mu)\kappa = (1 - \mu)\left(\frac{\mu(1-\mu)}{\sigma^2} - 1\right), \tag{6.7}$$

provided $\sigma^2 < \mu(1 - \mu)$.

6.3.3 beta_params()

`CalvinBayes::beta_params()` will compute several summaries of a beta distribution given any of these 2-parameter summaries. This can be very handy for converting from one type of information about a beta distribution to another.

For example. Suppose you want a beta distribution with mean 0.3 and standard deviation 0.1. Which beta distribution is it?

```
library(CalvinBayes)
beta_params(mean = 0.3, sd = 0.1)
```

shape1	shape2	mean	mode	sd	concentration
6	14	0.3	0.2778	0.1	20

We can do a similar thing with other combinations.

```
bind_rows(
  beta_params(mean = 0.3, concentration = 10),
  beta_params(mode = 0.3, concentration = 10),
  beta_params(mean = 0.3, sd = 0.2),
  beta_params(shape1 = 5, shape2 = 10),
)
```

shape1	shape2	mean	mode	sd	concentration
3.000	7.000	0.3000	0.2500	0.1382	10.00
3.400	6.600	0.3400	0.3000	0.1428	10.00
1.275	2.975	0.3000	0.1222	0.2000	4.25
5.000	10.000	0.3333	0.3077	0.1179	15.00

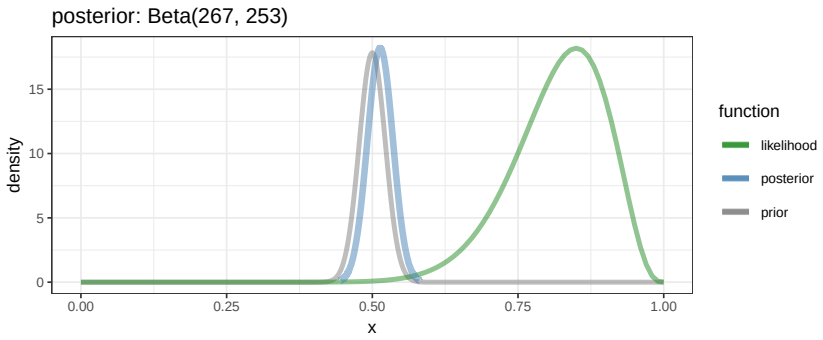
6.3.4 Automating Bayesian updates for a proportion (beta prior)

Since we have formulas for this case, we can write a function handle any beta prior and any data set very simply. (Much simpler than doing the grid method each time).

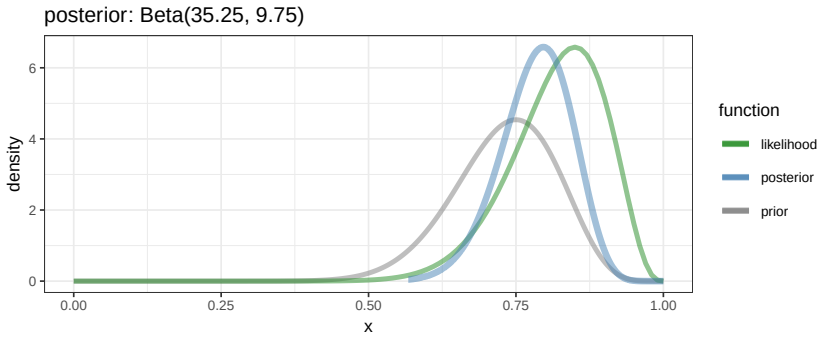
```
quick_bern_beta <-
  function(
    x, n,      # data, successes and trials
    ...        # see clever trick below
  )
  {
    pars <- beta_params(...)
    a <- pars$shape1
    b <- pars$shape2

    theta_hat <- x / n  # value that makes likelihood largest
    posterior_mode <- (a + x - 1) / (a + b + n - 2)

    # scale likelihood to be as tall as the posterior
    likelihood <- function(theta) {
      dbinom(x, n, theta) / dbinom(x, n, theta_hat) *
        dbeta(posterior_mode, a + x, b + n - x)  # posterior height at mode
    }

    gf_dist("beta", shape1 = a, shape2 = b,
            color = ~ "prior", alpha = 0.5, xlim = c(0,1), size = 1.2) %>%
      gf_function(likelihood,
                  color = ~ "likelihood", alpha = 0.5, size = 1.2) %>%
      gf_dist("beta", shape1 = a + x, shape2 = b + n - x,
              color = ~ "posterior", alpha = 0.5, size = 1.6) %>%
      gf_labs(
        color = "function",
        title = paste0("posterior: Beta(", a + x, ", ", b + n - x, ")")
      ) %>%
      gf_refine(
        scale_color_manual(
          values = c("prior" = "gray50", "likelihood" = "forestgreen",
                     "posterior" = "steelblue")))
  }
```

With such a function in hand, we can explore examples very quickly. Here are three examples from *DBDA2e* (pp. 134-135).

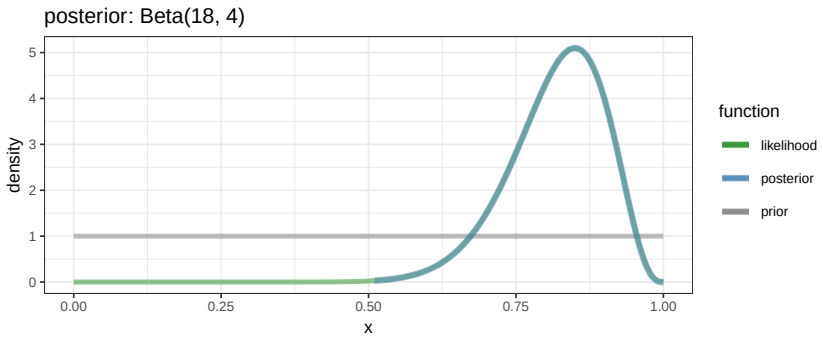
```
quick_bern_beta(17, 20, mode = 0.5, k = 500)
```




```
quick_bern_beta(17, 20, mode = 0.75, k = 25)
```




```
quick_bern_beta(17, 20, a = 1, b = 1)
```



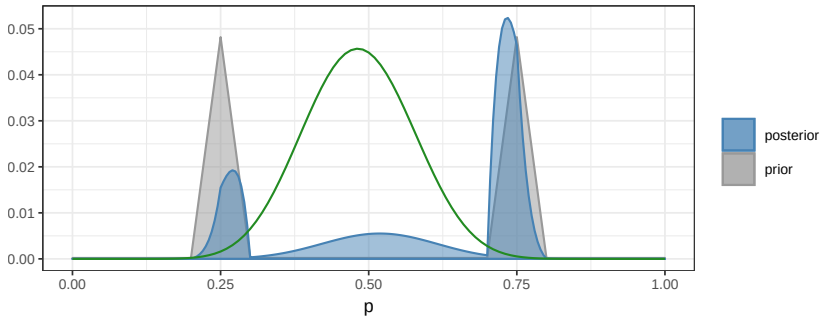
6.4 What if the prior isn't a beta distribution?

Unless it is some other distribution where we can work things out mathematically, we are back to the grid method.

Here's an example like the one on page 136.

```
dtwopeaks <- function(x) {
  0.48 * triangle::dtriangle(x, 0.2, 0.3) +
  0.48 * triangle::dtriangle(x, 0.7, 0.8) +
  0.04 * dunif(x)
}

BernGrid(data = c(rep(0, 13), rep(1, 14)), prior = dtwopeaks) %>%
  gf_function(function(theta) 0.3 * dbinom(13, 27, theta), color = "forestgreen")
```

```
13 + 14 = 27
```



6.5 Exercises

1. Show that if $\alpha, \beta > 1$, then the mode of a Beta(α, β) distribution is $\dfrac{\alpha - 1}{\alpha + \beta - 2}$.

   Hint: What would you do if you were in Calculus I?

2. Suppose we have a coin that we know comes from a magic-trick store, and therefore we believe that the coin is strongly biased either usually to come up heads or usually to come up tails, but we don't know which.

   a. Express this belief as a beta prior. That is, find shape parameters that lead to a beta distribution that corresponds to this belief.

   b. Now we flip the coin 5 times and it comes up heads in 4 of the 5 flips. What is the posterior distribution?

   c. Use `quick_bern_beta()` or a similar function of your own creation to show the prior and posterior graphically.

3. Suppose we estimate a proprtion θ using a Beta$(10, 10)$ prior and a observe 26 successes and 48 failures.

   a. What is the posterior distribution?
   b. What is the mean of the posterior distribution?
   c. What is the mode of the posterior distribution?
   d. Compute a 90% HDI for θ. [Hint: `qbeta()`]

4. Suppose a state-wide election is approaching, and you are interested in knowing whether the general population prefers the democrat or the republican. There is a just-published poll in the newspaper, which states that of 100 randomly sampled people, 58 preferred the republican and the remainder preferred the democrat.

   a. Suppose that before the newspaper poll, your prior belief was a uniform distribution. What is the 95% HDI on your beliefs after learning of the newspaper poll results?

   b. Based on what you know about elections, why is a uniform prior not a great choice? Repeat part (a) with a prior the conforms better to what you know about elections. How much does the change of prior affect the 95% HDI?

   c. You find another poll conducted by a different news organization In this second poll, 56 of 100 people preferred the republican. Assuming that peoples' opinions have not changed between polls, what is the 95% HDI on the posterior taking both polls into account. Make it clear which prior you are using.

   d. Based on this data (and your choice of prior, and assuming public opinion doesn't change between the time of the polls and election day), what is the probability that the republican will win the election.

Chapter 7

Markov Chain Monte Carlo (MCMC)

7.1 King Markov and Adviser Metropolis

King Markov is king of a chain of 5 islands. Rather than live in a palace, he lives in a royal boat. Each night the royal boat anchors in the harbor of one of the islands. The law declares that **the king must harbor at each island in proportion to the population of the island**.

Question 1: If the populations of the islands are 100, 200, 300, 400, and 500 people, how often must King Markov harbor at each island?

King Markov has some personality quirks:

- He can't stand record keeping. So he doesn't know the populations on his islands and doesn't keep track of which islands he has visited when.
- He can't stand routine (variety is the spice of his life), so he doesn't want to know each night where he will be the next night.

He asks Adviser Metropolis to devise a way for him to obey the law but that

- randomly picks which island to stay at each night,
- doesn't require him to remember where he has been in the past, and
- doesn't require him to remember the populations of all the islands.

He can ask the clerk on any island what the island's population is whenever he needs to know. But it takes half a day to sail from one island to another, so he is limited in how much information he can obtain this way each day.

Metropolis devises the following scheme:

- Each morning, have breakfast with the island clerk and inquire about the population of the current island.
- Then randomly pick one of the 4 other islands (a proposal island) and travel there in the morning
  - Let $J(b \mid a)$ be the conditional probability of selecting island b as the candidate if a is the current island.
  - J does not depend on the populations of the islands (since the King can't remember them).
- Over lunch at the proposal island, inquire about its population.
  - If the proposal island has more people, stay at the proposal island for the night (since the king should prefer more populated islands).
  - If the proposal island has fewer people, stay at the proposal island with probability A, else return to the "current" island (ie, last night's island).

Metropolis is convinced that for the right choices of J and A, this will satisfy the law.

He quickly determines that A cannot be 0 and cannot be 1:

Question 2. What happens if $A = 0$? What happens if $A = 1$?

It seems like A might need to depend on the populations of the current and proposal islands. When we want to emphasize that, we'll denote it as $A = A(b \mid a)$. But how? If A is too large, the king will visit small islands too often. If A is too small, he will visit large islands too often.

Fortunately, Metropolis knows about Markov Chains. Unfortunately, some of you may not. So let's learn a little bit about Markov Chains and then figure out how Metropolis should choose J and A.

7.2 Quick Intro to Markov Chains

7.2.1 More info, please

This is going to be very quick. You can learn more, if you are interested, by going to

- https://www.dartmouth.edu/~chance/teaching_aids/books_articles/probability_book/Chapter11.pdf

7.2.2 Definition

Consider a random process that proceeds in discrete steps (often referred to as time). Let X_t represent the "state" of the process at time t. Since this is a random process, X_t is random, and we can ask probability questions like "What is the probability of being in state _____ at time _____?", ie, What is $\Pr(X_t = x)$?

If

$$\Pr(X_{t+1} = x \mid X_t = x_t, X_{t-1} = x_{t-1}, \ldots, X_0 = x_0) = \Pr(X_{t+1} \mid X_t = x_t)$$

then we say that the process is a **Markov Chain**. The intuition is that (the probabilities of) what happens next depends only on the current state and not on previous history.

7.2.3 Time-Homogeneous Markov Chains

The simplest version of a Markov Chain is one that is **time-homogeneous**:

$$\Pr(X_{t+1} = b \mid X_t = a) = p_{ab}$$

That is, the (conditional) probability of moving from state a to state b in one step is the same at every time.

7.2.4 Matrix representation

A time-homogeneous Markov Chain can be represented by a square matrix M with

$$M_{ij} = p_{ij} = \text{probability of transition from state } i \text{ to state } j \text{ in one step}$$

(This will be an infinite matrix if the state space in infinite, but we'll start with simple examples with small, finite state spaces.) M_{ij} is the probability of moving in one step from state i to state j.

More generally, we will write $M_{ij}^{(k)}$ for the probability of moving from state i to state j in k steps.

Small Example:

```
M <- rbind( c(0, 0.5, 0.5), c(0.25, 0.25, 0.5), c(0.5, 0.3, 0.2))
M
```

0.00	0.50	0.5
0.25	0.25	0.5
0.50	0.30	0.2

Question 3:

- How many states does this process have?
- What is the probability of moving from state 1 to state 3 in 1 step?
- What is the probability of moving from state 1 to state 3 in 2 steps? (Hint: what are the possible stopping points along the way?)
- How do we obtain $M^{(2)}$ from M?
- How do we obtain $M^{(k)}$ from M?

Question 4: The Metropolis Algorithm as a Markov process

- What are the states of the Metropolis algorithm?
- If King Markov is on island 2, what is the probability of moving to Island 3?
- If King Markov is on island 3, what is the probability of moving to Island 2?
- What is the general formula for the probability of moving from island a to island b (in one step)? ($\Pr(X_{t+1} = b \mid X_t = a)$)

7.2.5 Regular Markov Chains

A time-homogeneous Markov Chain, is called **regular** if there is a number k such that

- every state is reachable from every other state with non-zero probability in k steps

Question 5a

- Is our small example regular? If so, how many steps are required?

Question 5b

- Under what conditions is the Metropolis algorithm regular?

Regular Markov Chains have a very nice property:

$$\lim_{k \to \infty} M^{(k)} = W$$

where every row of W is the same. This says that, no matter where you start the process, the long-run probability of being in each state will be the same.

In our small example above, convergence is quite rapid:

```
M %^% 20
```

0.2769	0.3385	0.3846
0.2769	0.3385	0.3846
0.2769	0.3385	0.3846

```
M %^% 21
```

0.2769	0.3385	0.3846
0.2769	0.3385	0.3846
0.2769	0.3385	0.3846

Note: If we apply the matrix M to the limiting probability (w, one row of W), we just get w back again:

$$wM = w$$

```
W <- M %^% 30
W[1,]
```

```
## [1] 0.2769 0.3385 0.3846
```

```
W[1,] %*% M
```

0.2769	0.3385	0.3846

In fact, this is a necessary and sufficient condition for the limiting probability.

So, here's what Metropolis needs to do: Choose J and A so that

- his algorithm is a regular Markov Chain with matrix M
- If $w = \langle p(1), p(2), p(3), p(4), p(5) \rangle$ is the law-prescribed probabilities for island harboring, then $wM = w$.

7.3 Back to King Markov

If A is between 0 and 1, and the jumping rule allows us to get to all the islands (eventually), then the Markov Chain will be regular, so there will be a limiting distribution. But the limiting distribution must be the one the law requires. It suffices to show that if the law is satisfied at time t it is satisfied at time $t+1$ ($wM = w$):

$$\Pr(X_t = a) = p(a) \text{ for all } a \Rightarrow \Pr(X_{t+1} = a) = p(a) \text{ for all } a$$

Here's the trick: We will choose J and A so that the following two **unconditional** probabilities are equal.

$$\Pr(a \rightarrow_t b) = \Pr(b \rightarrow_t a)$$

where $\Pr(a \rightarrow_t b) = \Pr(X_t = a \ \& \ X_{t+1} = b)$.

Why does this work?

- Suppose $\Pr(X_t = a) = p(a)$ as the law prescribes.
- $\Pr(a \rightarrow_t b) = \Pr(b \rightarrow_t a)$ makes the joint distribution symmetric: For any a and any b.

$$\Pr(X_t = a \ \& \ X_{t+1} = b) = \Pr(X_t = b \ \& \ X_{t+1} = a)$$

- This means that both marginals are the same, so for any a:

$$\Pr(X_t = a) = \Pr(X_{t+1} = a)$$

- In other words, the probability of the current island will be the same as the probability of the next island: $wM = w$.

Time for some algebra (and probability)! How do we choose J and A? Recall the ingredients:

- $P(a)$ be the population of island a
- $p(a)$ be the proportion of the total population living on island a: $p(a) = \frac{p(a)}{\sum_x p(x)}$
- $J(b \mid a)$ is the conditional probability of selecting island b as the candidate when a is the current island. (J for Jump probability)
- $A(b \mid a)$ is the probability of accepting proposal island b if it is proposed from island a.

Question 6: Consider two islands – a and b – with $P(b) > P(a)$. Assume that probability of being on island x is $p(x)$. Calculate the following probabilities (in terms of things like p, J, and A).

- (Unconditional) probability of moving from a to $b = \Pr(a \to_t b) =$
- (Unconditional) probability of moving from b to $a = \Pr(b \to_t a) =$

Question 7: How do we choose J and A to make these probabilities equal?

$$A(a \mid b) =$$

Question 8: Symmetric jump rules.

- Is it possible to do this with symmetric jump rules? That is, can we require $J(b \mid a) = J(a \mid b)$? (Remember, the king doesn't like to remember stuff, and this means half as much stuff to remember about the jump rules).
- Does using a symmetric jump rule make the acceptance rule A any simpler or more complicated? (The king won't be so happy if the simpler jump rule makes the acceptance rule a lot more complicated.)

Question 9: Constant jump rules.

- Is it possible to do this if we require that $J(y \mid x) = J(y' \mid x')$ for all $y \neq x$ and $y' \neq x'$? (This would make life even easier for the king.)
- For King Markov, what would J be if we did it this way?

The original Metropolis algorithm used symmetric jump rules. The later generalization (Metropolis-Hastings) employed non-symmetric jump rules to get better performance of the Markov Chain.

7.4 How well does the Metropolis Algorithm work?

Let's let the computer simulate this algorithm. And since the computer is doing all the work, let's make a general function so we can experiment a bit.

```
KingMarkov <- function(
  num_steps    = 1e5,
  population   = 1:5,
  island_names = 1:length(population),
  start        = 1,
  J = function(a, b) {1 / (length(population) - 1)}
  ) {

  num_islands  <- length(population)
  island_seq   <- rep(NA, num_steps)  # trick to pre-alocate memory
  proposal_seq <- rep(NA, num_steps)  # trick to pre-alocate memory
  current      <- start
  proposal     <- NA

  for (i in 1:num_steps) {
    # record current island
    island_seq[i]   <- current
    proposal_seq[i] <- proposal

    # propose one of the other islands
    other_islands <- setdiff(1:num_islands, current)
    proposal <-
```

```
      sample(other_islands, 1,
             prob = purrr::map(other_islands, ~ J(current, .x)))
    # move?
    prob_move <- population[proposal] / population[current]
    # new current island (either current current or proposal)
    current   <- ifelse(runif(1) < prob_move, proposal, current)
  }
  tibble(
    step     = 1:num_steps,
    island   = island_names[island_seq],
    proposal = island_names[proposal_seq]
  )
}
```

Question 10: Look at the code above and answer the following.

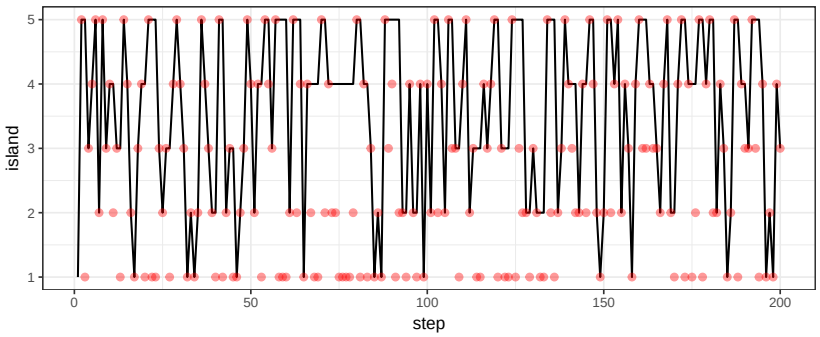
- What are the default populations of the islands?
- What is the default jump rule?
- Explain what each of the following bits of code are doing:
  - `other_islands <- setdiff(1:num_islands, current)`
  - `prob = purrr::map(other_islands, ~ J(current, .x))`
  - the call to `sample()`
  - `prob_move <- population[proposal] / population[current]`
  - `current   <- ifelse(runif(1) < prob_move, proposal, current)`
  - the call to `tibble()`

7.4.1 Jumping to any island

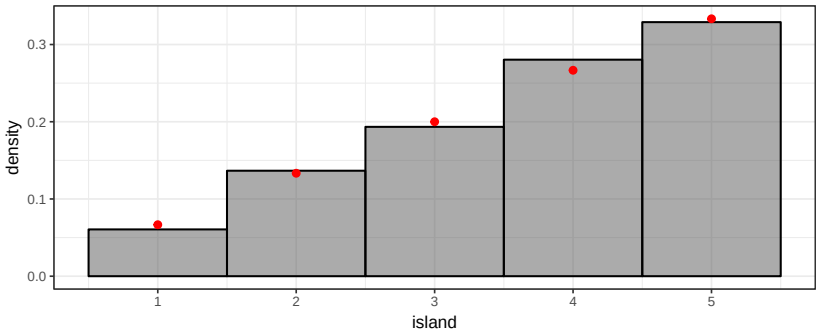
Now let's simulate the first 5000 nights of King Markov's reign.

```
Tour <- KingMarkov(5000)
Target <- tibble(island = 1:5, prop = (1:5)/ sum(1:5))
gf_line(island ~ step, data = Tour %>% filter(step <= 200)) %>%
  gf_point(proposal ~ step, data = Tour %>% filter(step <= 200),
           color = "red", alpha = 0.4) %>%
  gf_refine(scale_y_continuous(breaks = 1:10))
```

```
## Warning: Removed 1 rows containing missing values (geom_point).
```




```
gf_dhistogram( ~ island, data = Tour, binwidth = 1, color = "black") %>%
  gf_point(prop ~ island, data = Target, color = "red") %>%
  gf_refine(scale_x_continuous(breaks = 1:10))
```



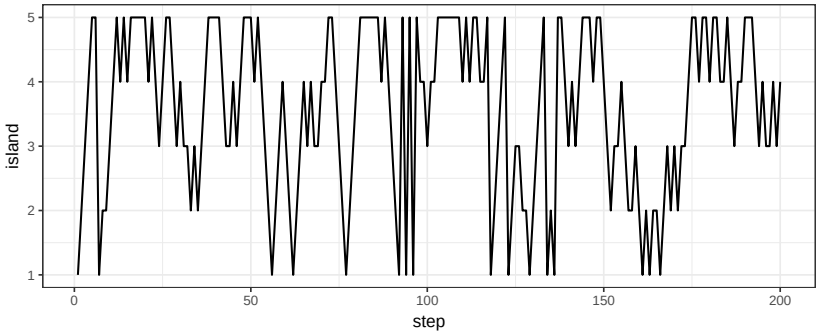
Question 11: Look at the first plot. It shows where the king stayed each of the first 200 nights.

- Did the king ever stay two consecutive nights on the same island? (How can you tell from the plot?)
- Did the king ever stay two consecutive nights on the smallest island? (How can you answer this without looking at the plot?)

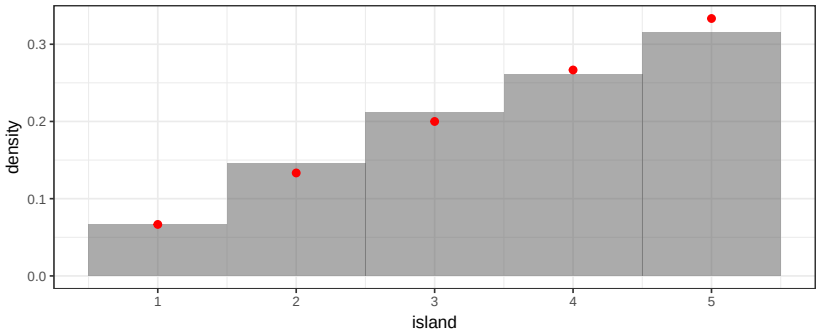
7.4.2 Jumping only to neighbor islands

What if we only allow jumping to neighboring islands? (Imagine the islands are arranged in a circle and we only sail clockwise or counterclockwise around the circle to the nearest island.)

```
neighbor <- function(a, b) as.numeric(abs(a-b) %in% c(1,4))
Tour <- KingMarkov(10000, population = 1:5, J = neighbor)
Target <- tibble(island = 1:5, prop = (1:5)/ sum(1:5))
gf_line(island ~ step, data = Tour %>% filter(step <= 200)) %>%
  gf_refine(scale_y_continuous(breaks = 1:10))
```

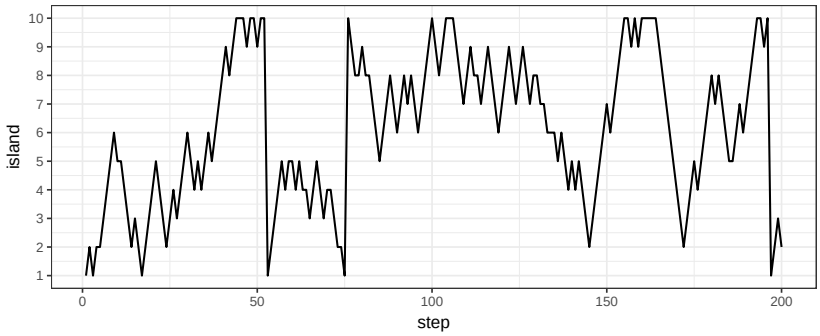



```
gf_dhistogram( ~ island, data = Tour, binwidth = 1) %>%
  gf_point(prop ~ island, data = Target, color = "red") %>%
  gf_refine(scale_x_continuous(breaks = 1:10))
```

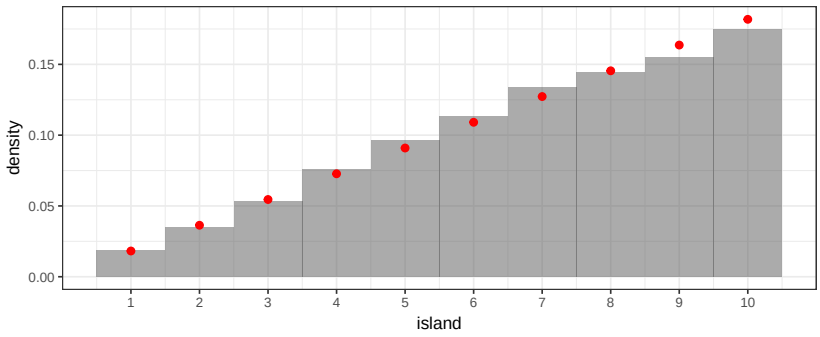


The effect of visiting only neighbors is more dramatic with more islands.

```
neighbor <- function(a, b) as.numeric(abs(a-b) %in% c(1,9))
Tour <- KingMarkov(10000, population = 1:10, J = neighbor)
Target <- tibble(island = 1:10, prop = (1:10)/ sum(1:10))
gf_line(island ~ step, data = Tour %>% filter(step <= 200)) %>%
  gf_refine(scale_y_continuous(breaks = 1:10))
```




```
gf_dhistogram( ~ island, data = Tour, binwidth = 1) %>%
  gf_point(prop ~ island, data = Target, color = "red") %>%
  gf_refine(scale_x_continuous(breaks = 1:10))
```



7.5 Markov Chains and Posterior Sampling

That was a nice story, and some nice probability theory. But what does it have to do with Bayesian computation? Regular Markov Chains (and some generalizations of them) can be used to sample from a posterior distribution:

- state = island = set of parameter values
  - in typical applications, this will be an infinite state space
- population = prior * likelihood
  - importantly, we do not need to normalize the posterior; that would typically be a very computationally expensive thing to do
- start in any island = start at any parameter values
  - convergence may be faster from some starting states than from others, but in principle, any state will do
- randomly choose a proposal island = randomly select a proposal set of parameter values
  - if the posterior is greater there, move
  - if the posterior is smaller, move anyway with probability

$$A = \frac{\text{proposal 'posterior'}}{\text{current 'posterior'}}$$

Metropolis-Hastings variation:

- More choices for $J()$ (need not be symmetric) gives more opportunity to tune for convergence

Other variations:

- Can allow M to change over the course of the algorithm. (No longer time-homogeneous.)

7.5.1 Example 1: Estimating a proportion

To see how this works in practice, let's consider our familiar model that has a Bernoulli likelihood and a beta prior:

- $Y_i \sim \text{Bern}(\theta)$
- $\theta \sim \text{Beta}(a, b)$

Since we are already familiar with situation, we know that the posterior should be a beta distribution when the prior is a beta distribution. We can use this information to see how well the algorithm works in that situation.

Let's code up our Metropolis algorithm for this situation. There is a new wrinkle, however: The state space for the parameter is a continuous interval [0,1]. So we need a new kind of jump rule

- Instead of sampling from a finite state space, we use `rnorm()`
- The standard deviation of the normal distribution (called `size` in the code below) controls how large a step we take (on average). This number has nothing to do with the model, it is a **tuning parameter** of the algorithm.

```
metro_bern <- function(
  x, n,            # x = successes, n = trials
  size = 0.01,     # sd of jump distribution
  start = 0.5,     # value of theta to start at
  num_steps = 1e4, # number of steps to run the algorithm
  prior = dunif,   # function describing prior
  ...              # additional arguments for prior
  ) {
```

```
  theta             <- rep(NA, num_steps)  # trick to pre-alocate memory
  proposed_theta    <- rep(NA, num_steps)  # trick to pre-alocate memory
  move              <- rep(NA, num_steps)  # trick to pre-alocate memory
  theta[1]          <- start

  for (i in 1:(num_steps-1)) {
    # head to new "island"
    proposed_theta[i + 1] <- rnorm(1, theta[i], size)

    if (proposed_theta[i + 1] <= 0 ||
        proposed_theta[i + 1] >= 1) {
      prob_move <- 0           # because prior is 0
    } else {
      current_prior       <- prior(theta[i], ...)
      current_likelihood  <- dbinom(x, n, theta[i])
      current_posterior   <- current_prior * current_likelihood
      proposed_prior      <- prior(proposed_theta[i+1], ...)
      proposed_likelihood <- dbinom(x, n, proposed_theta[i+1])
      proposed_posterior  <- proposed_prior * proposed_likelihood
      prob_move           <- proposed_posterior / current_posterior
    }


    # sometimes we "sail back"
    if (runif(1) > prob_move) { # sail back
       move[i + 1] <- FALSE
      theta[i + 1] <- theta[i]
    } else {                    # stay
       move[i + 1] <- TRUE
      theta[i + 1] <- proposed_theta[i + 1]
    }
  }

  tibble(
    step = 1:num_steps,
    theta = theta,
    proposed_theta = proposed_theta,
    move = move, size = size
  )
}
```

Question 12: What happens if the proposed value for θ is not in the interval $[0, 1]$? Why?

Question 13: What do `proposed_posterior` and `current_posterior` correspond to in the story of King Markov?

Question 14: Notice that we are using the unnormalized posterior. Why don't we need to normalize the posterior? Why is it important that we don't have to normalize the posterior?

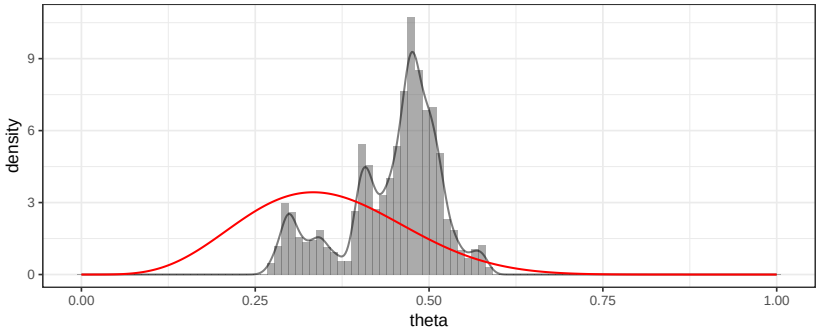
7.5.1.1 Looking at posterior samples

The purpose of all this was to get samples from the posterior distribution. We can use histograms or density plots to see what our MCMC algorithm shows us for the posterior distribution. When using MCMC algorithms, we won't typically have ways of knowing the "right answer". But in this case, we know the

posterior is Beta(6, 11), so we can compare our posterior samples to that distribution to see how well things worked.

Let's try a nice small step size like 0.2%.

```
set.seed(341)
Tour <- metro_bern(5, 15, size = 0.002)
gf_dhistogram(~ theta, data = Tour, bins = 100) %>%
  gf_dens(~ theta, data = Tour) %>%
  gf_dist("beta", shape1 = 6, shape2 = 11, color = "red")
```

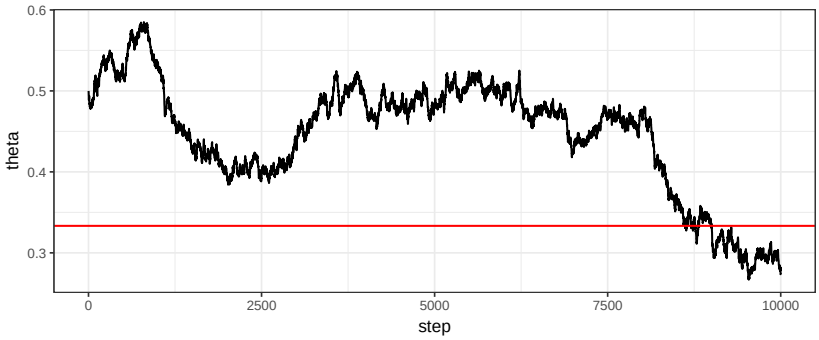


Hmm. That's not too good. Let's see if we can figure out why.

7.5.1.2 Trace Plots

A trace plot shows the "tour of King Markov's ship" – the sequence of parameter values sampled (in order).

```
gf_line(theta ~ step, data = Tour) %>%
  gf_hline(yintercept = 5/15, color = "red")
```



Question 15: Why does the trace plot begin at a height of 0.5?

Question 16: What features of this trace plot indicate that our posterior sampling isn't working well? Would making the step size larger or smaller be more likely to help?

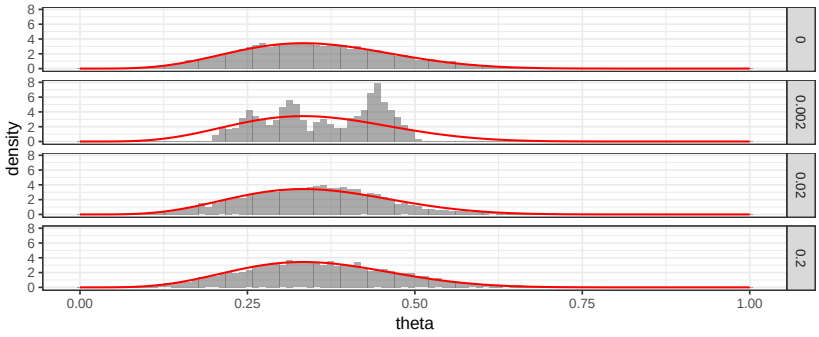
Question 17: Since we know that the posterior distribution is Beta(6, 11), how could we use R to show us what an ideal trace plot would look like?

7.5.1.3 Comparing step sizes

Let's see how our choice of `step` affects the sampling. Size 0 is sampling from a true Beta(6, 11) distribution.

```
set.seed(341)
Tours <-
  bind_rows(
    metro_bern(5, 15, size = 0.02),
    metro_bern(5, 15, size = 0.2),
    metro_bern(5, 15, size = 0.002),
    tibble(theta = rbeta(1e4, 6, 11), size = 0, step = 1:1e4)
  )
```

```
gf_dhistogram( ~ theta | size ~ ., data = Tours, bins = 100) %>%
  gf_dist("beta", shape1 = 6, shape2 = 11, color = "red")
```



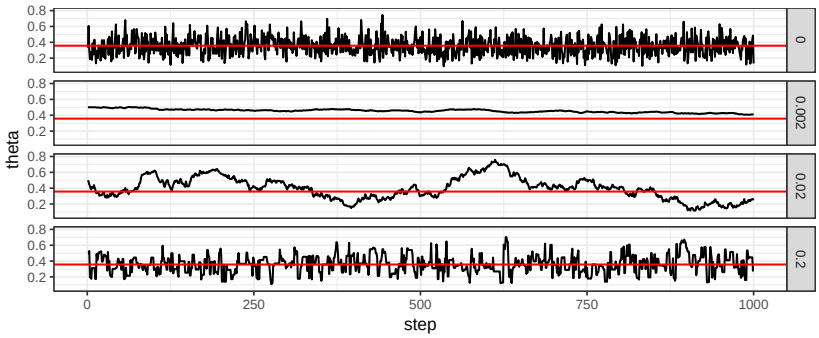
Sizes 0.2 and 0.02 look much better than 0.002.

Looking at the trace plots, we see that the samples using a step size of 0.02 are still highly auto-correlated (neighboring values are similar to each other). Thus the "effective sample size" is not nearly as big as the number of posterior samples we have generated.

With a step size of 0.2, the amount of auto-correlation is substantially reduced, but not eliminated.

```
gf_line(theta ~ step, data = Tours) %>%
  gf_hline(yintercept = 5/14, color = "red") %>%
  gf_facet_grid(size ~ .) %>%
  gf_lims(x = c(0,1000))
```

```
## Warning: Removed 9000 rows containing missing values (geom_path).
```

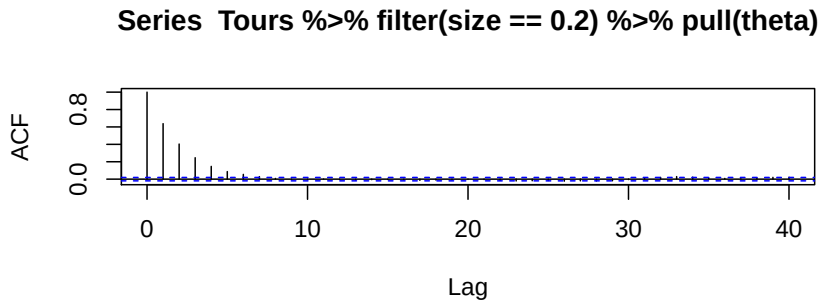


7.5.1.4 Auto-correlation and Thinning

One way to reduce the auto-correlation in posterior samples is by thinning. This auto-correlation plot suggests that keeping every 7th or 8th value when `size` is 0.2 should give us a posterior sample that is nearly

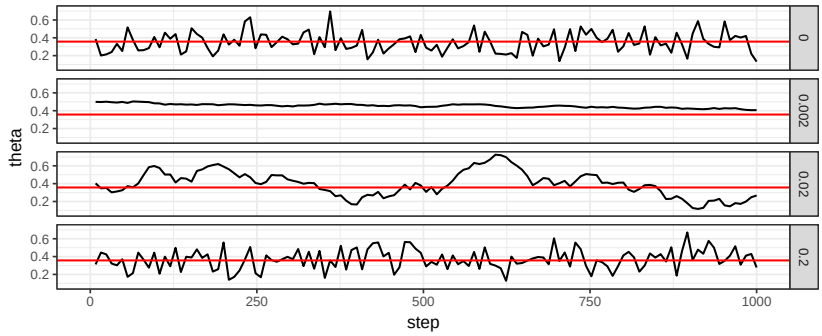
independent.

```
acf(Tours %>% filter(size == 0.2) %>% pull(theta))
```




```
gf_line(theta ~ step, data = Tours %>% filter(step %% 8 == 0)) %>%
  gf_hline(yintercept = 5/14, color = "red") %>%
  gf_facet_grid(size ~ .) %>%
  gf_lims(x = c(0,1000))
```

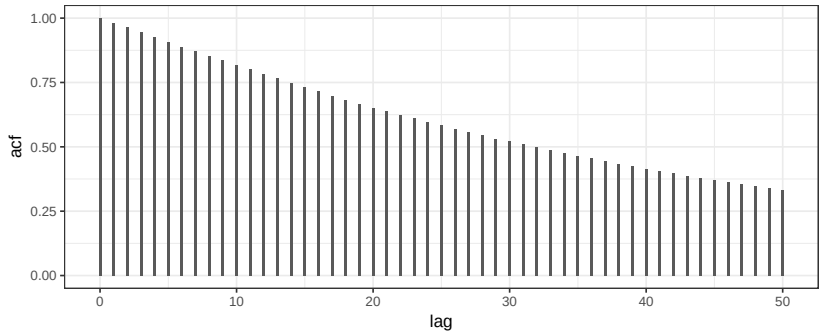
```
## Warning: Removed 1125 rows containing missing values (geom_path).
```



Now the idealized trace plot and the trace plot for `step = 0.2` are quite similar looking. So the effective sample size for that step size is approximately 1/8 the number of posterior samples generated.

For `step=0.02` we must thin even more, which would require generating many more posterior samples to begin with:

```
ACF <-
  broom::tidy(acf(Tours %>% filter(size == 0.02, step > 1000) %>% pull(theta),
                  plot = FALSE, lag.max = 50))
gf_col(acf ~ lag, data = ACF, width = 0.2)
```



7.5.2 Example 2: Estimating mean and variance

Consider the following simple model:

- $Y_i \sim \text{Norm}(\mu, \sigma)$
- $\mu \sim \text{Norm}(0, 1)$
- $\log(\sigma) \sim \text{Norm}(0, 1)$

In this case the posterior distribution for μ can be worked out exactly and should be normal.

Let's code up our Metropolis algorithm for this situation. New stuff:

- we have two parameters, so we'll use separate jumps for each and combine
  - we could use a jump rule based on both values together, but we'll keep this simple
- the state space for each parameter is infinite, so we need a new kind of jump rule
  - instead of sampling from a finite state space, we use `rnorm()`
  - the standard deviation controls how large a step we take (on average)
  - example below uses same standard deviation for both parameters, but we should select them individually if the parameters are on different scales

```
metro_norm <- function(
  y,                # data vector
  num_steps = 1e5,
  size = 1,         # sd's of jump distributions
  start = list(mu = 0, log_sigma = 0)
  ) {

  size <- rep(size, 2)[1:2]        # make sure exactly two values
  mu        <- rep(NA, num_steps)  # trick to pre-alocate memory
  log_sigma <- rep(NA, num_steps)  # trick to pre-alocate memory
  move      <- rep(NA, num_steps)  # trick to pre-alocate memory
  mu[1] <- start$mu
  log_sigma[1] <- start$log_sigma
  move[1] <- TRUE

  for (i in 1:(num_steps - 1)) {
    # head to new "island"
    mu[i + 1]        <- rnorm(1, mu[i], size[1])
    log_sigma[i + 1] <- rnorm(1, log_sigma[i], size[2])
    move[i + 1] <- TRUE

    log_post_current <-
      dnorm(mu[i], 0, 1, log = TRUE) +
      dnorm(log_sigma[i], 0, 1, log = TRUE) +
      sum(dnorm(y, mu[i], exp(log_sigma[i]), log = TRUE))
    log_post_proposal <-
      dnorm(mu[i + 1], 0, 1, log = TRUE) +
      dnorm(log_sigma[i + 1], 0, 1, log = TRUE) +
      sum(dnorm(y, mu[i + 1], exp(log_sigma[i+1]), log = TRUE))
    prob_move <- exp(log_post_proposal - log_post_current)

    # sometimes we "sail back"
    if (runif(1) > prob_move) {
      move[i + 1] <- FALSE
      mu[i + 1] <- mu[i]
      log_sigma[i + 1] <- log_sigma[i]
    }
```

```
  }
  tibble(
    step = 1:num_steps,
    mu = mu,
    log_sigma = log_sigma,
    move = move,
    size = paste(size, collapse = ", ")
  )
}
```

Let's use the algorithm with three different size values and compare results.

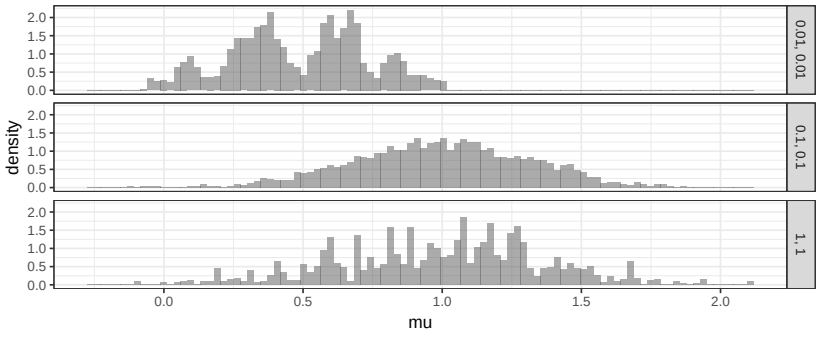
```
set.seed(341)
y <- rnorm(25, 1, 2)  # sample of 25 from Norm(1, 2)
Tour1    <- metro_norm(y = y, num_steps = 5000, size = 1)
Tour0.1  <- metro_norm(y = y, num_steps = 5000, size = 0.1)
Tour0.01 <- metro_norm(y = y, num_steps = 5000, size = 0.01)
Norm_Tours <-  bind_rows(Tour1, Tour0.1, Tour0.01)
```

```
df_stats(~ move | size, data = Norm_Tours, props)
```

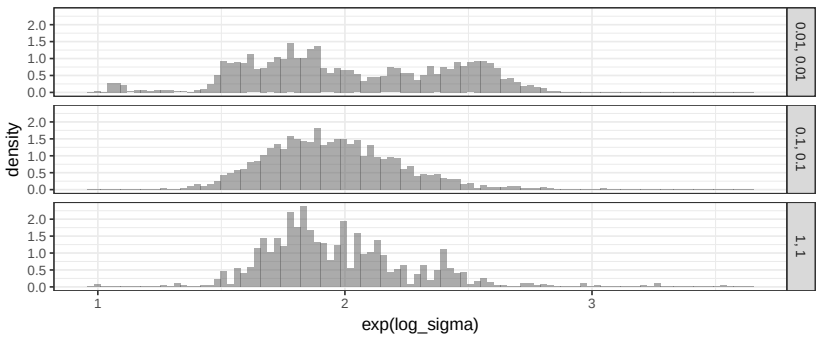
size	prop_FALSE	prop_TRUE
0.01, 0.01	0.0422	0.9578
0.1, 0.1	0.2380	0.7620
1, 1	0.9134	0.0866

7.5.2.1 Density plots

```
gf_dhistogram( ~ mu | size ~ ., data = Norm_Tours, bins = 100)
```

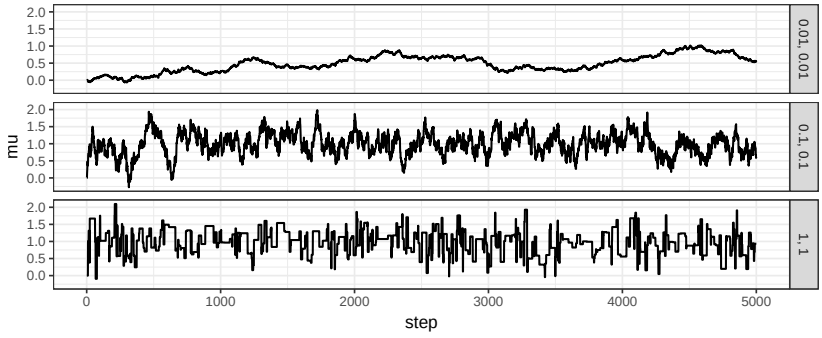



```
gf_dhistogram( ~ exp(log_sigma) | size ~ ., data = Norm_Tours, bins = 100)
```

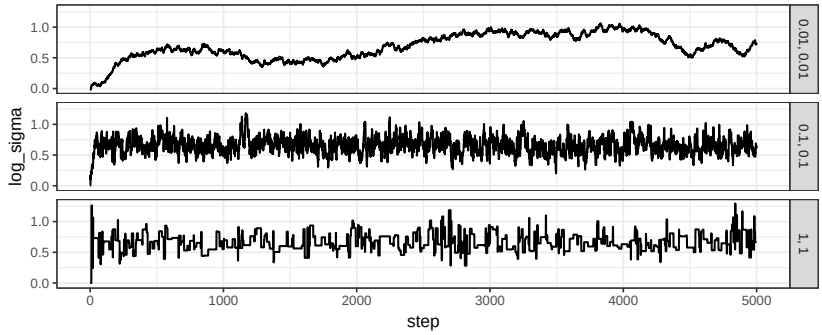


7.5.2.2 Trace plots

```
gf_line(mu ~ step | size ~ ., data = Norm_Tours)
```




```
gf_line(log_sigma ~ step | size ~ ., data = Norm_Tours)
```

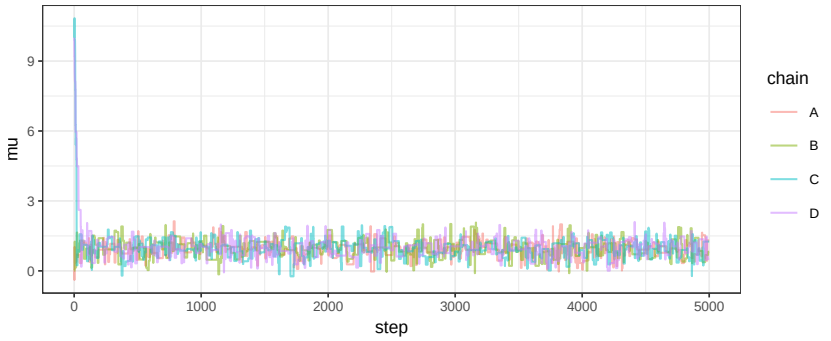


7.5.2.3 Comparing Multiple Chains

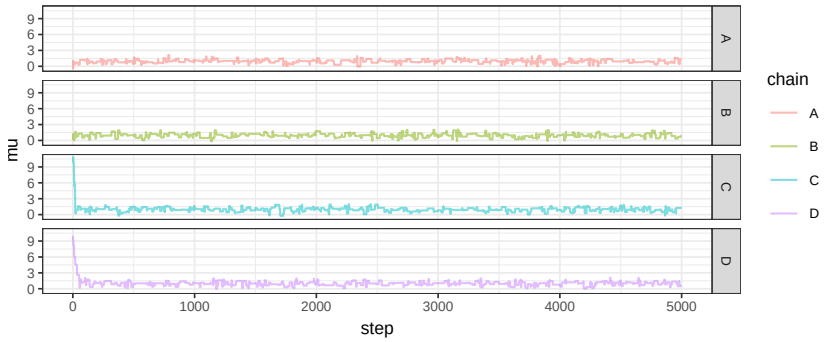
If we run multiple chains with different starting points and different random choices, we hope to see similar trace plots. After all, we don't want our analysis to be an analysis of starting points or of random choices.

```
Tour1a <- metro_norm(y = y, num_steps = 5000, size = 1) %>% mutate(chain = "A")
Tour1b <- metro_norm(y = y, num_steps = 5000, size = 1) %>% mutate(chain = "B")
Tour1c <- metro_norm(y = y, num_steps = 5000, size = 1, start = list(mu = 10, log_sigma = 5)) %>% mutat
Tour1d <- metro_norm(y = y, num_steps = 5000, size = 1, start = list(mu = 10, log_sigma = 5)) %>% mutat
Tours1 <- bind_rows(Tour1a, Tour1b, Tour1c, Tour1d)
```

```
gf_line(mu ~ step, color = ~chain, alpha = 0.5, data = Tours1)
```



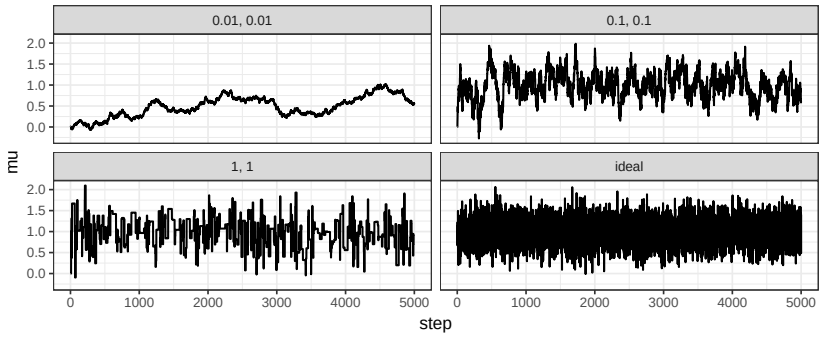

```
gf_line(mu ~ step, color = ~chain, alpha = 0.5, data = Tours1) %>%
  gf_facet_grid( chain ~ .)
```



7.5.2.4 Comparing Chains to an Ideal Chain

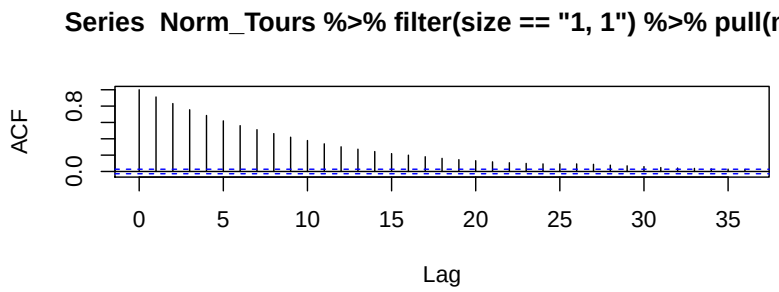
Not all posteriors are normal, but here's what a chain would look like if the posterior is normal and there is no correlation between draws.

```
Ideal <- tibble(step = 1:5000, mu = rnorm(5000, 1, .3), size = "ideal")
gf_line(mu ~ step | size, data = Norm_Tours %>% bind_rows(Ideal))
```

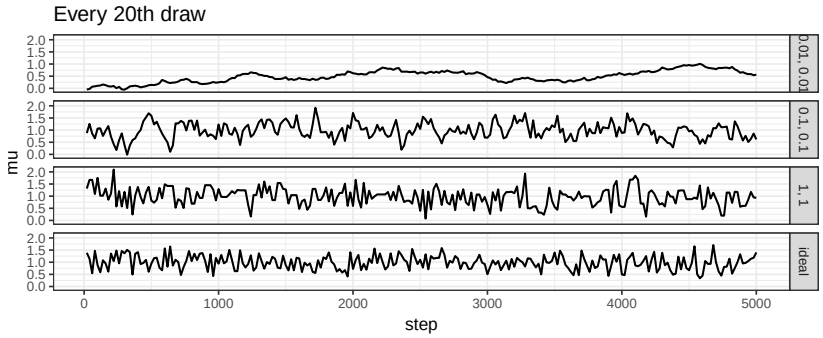


If the draws are correlated, then we might get more ideal behavior if we selected only a subset – every 20th or every 30th value, for example. This is the idea behind "effective sample size". The effective sample size of a correlated chain is the length of an ideal chain that contains as much independent information as the correlated chain.

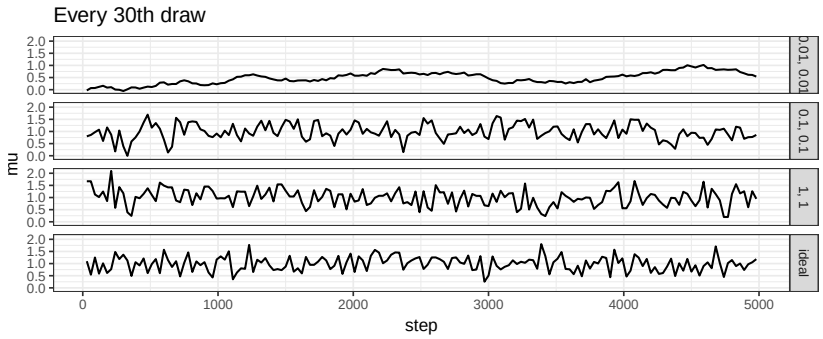
```
acf(Norm_Tours %>% filter(size == "1, 1") %>% pull(mu))
```




```
gf_line(mu ~ step | size ~ ., data = bind_rows(Norm_Tours, Ideal) %>% filter(step %% 20 == 0)) %>%
  gf_labs(title = "Every 20th draw")
```




```
gf_line(mu ~ step | size ~ ., data = bind_rows(Norm_Tours, Ideal) %>% filter(step %% 30 == 0)) %>%
  gf_labs(title = "Every 30th draw")
```

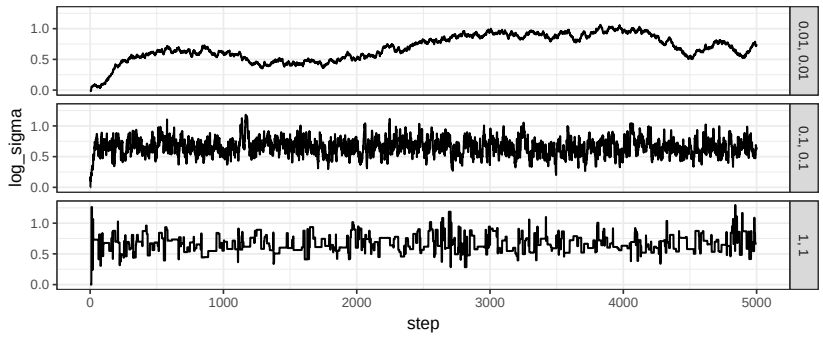


If we thin to every 20th value, our chains with `size = 1` and `size = 0.1` look quite similar to the ideal chain. The chain with `size = 0.01` still moves too slowly through the parameter space. So tuning parameters will affect the effective sample size.

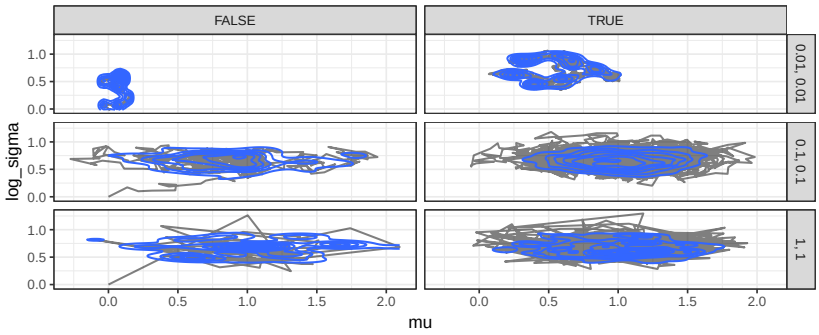
7.5.2.5 Discarding the first portion of a chain

The first portion of a chain may not work as well. This portion is typically removed from the analysis since it is more an indication of the starting values used than the long-run sampling from the posterior.

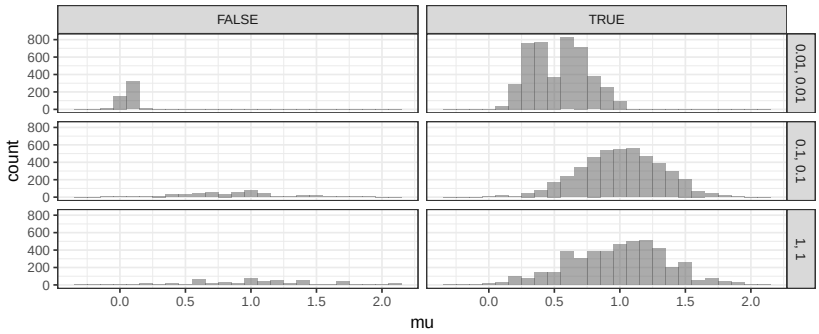
```
gf_line(log_sigma ~ step | size ~ ., data = Norm_Tours)
```



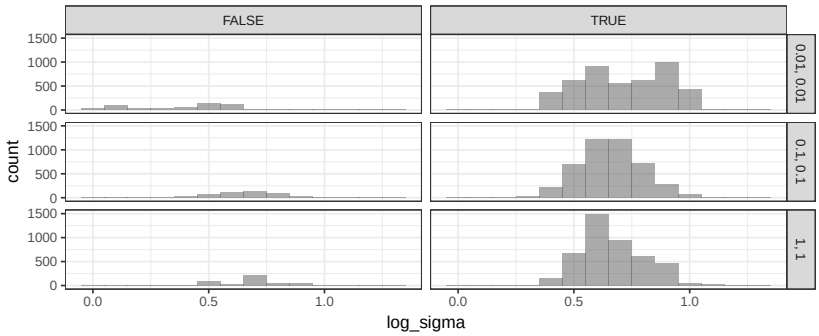

```
gf_path(log_sigma ~ mu | size ~ (step > 500), data = Norm_Tours, alpha = 0.5) %>%
  gf_density2d(log_sigma ~ mu, data = Norm_Tours)
```




```
gf_histogram( ~ mu | size ~ (step > 500) , data = Norm_Tours, binwidth = 0.1)
```




```
gf_histogram( ~ log_sigma | size ~ (step > 500), data = Norm_Tours, binwidth = 0.1)
```



7.5.3 Issues with Metropolis Algorithm

These are really issues with all MCMC algorithms, not just the Metropolis version:

- First portion of a chain might not be very good, need to discard it
- Tuning can affect performance – how do we tune?
- Samples are correlated – although the long-run probabilities are right, the next stop is not independent of the current one
  - so our effective posterior sample size isn't as big as it appears

7.6 Two coins

7.6.1 The model

Suppose we have two coins and want to compare the proportion of heads each coin generates.

- Parameters: θ_1, θ_2

- Data: N_1 flips of coin 1 and N_2 flips of coin 2. (Results: z_1 and z_2 heads, respectivly.)
- Likelihood: indepdendent Bernoulli (each flip independent of the other flips, each coin independent of the other coin)
- Prior: Independent Beta distributions for each θ_i

That is,

- $y_{1i} \sim \mathsf{Bern}(\theta_1), y_{2i} \sim \mathsf{Bern}(\theta_2)$
- $\theta_1 \sim \mathsf{Beta}(\mathsf{a}_1, \mathsf{b}_1), \theta_2 \sim \mathsf{Beta}(\mathsf{a}_2, \mathsf{b}_2)$ (independent)

For the examples below we will use data showing that 6 of 8 tosses of the first coin were heads and only 2 of 7 tosses of the second coin. But the methods work equally well with other data sets. While comparing a small number of coin tosses like this is not so interesting, the method can be used for a wide range of practical things, liking testing whether two treatments for a condition are equally effective, etc.

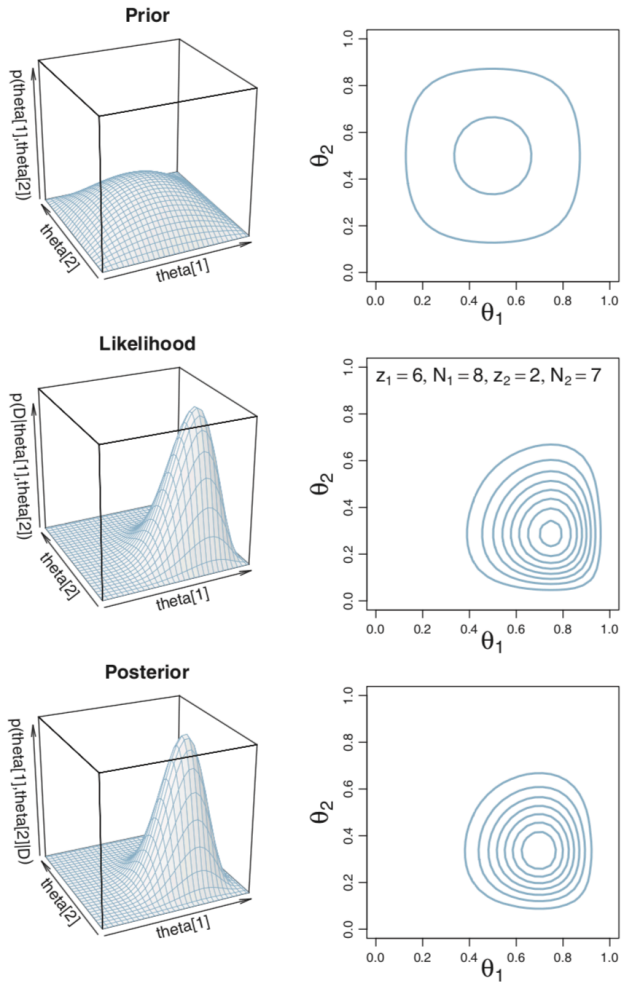
7.6.2 Exact analysis

From this we can work out the posterior as usual. (Pay attention to the important parts of the kernel, that is, the numerators.)

$$\begin{aligned}
p(\theta_1, \theta_2 \mid D) &= p(D \mid \theta_1, \theta_2) p(\theta_1, \theta_2) / p(D) \\
&= \theta_1^{z_1} (1\theta_1)^{N_1 z_1} \theta_1^{z_2} (1\theta_2)^{N_2 z_2} p(\theta_1, \theta_2) / p(D) \\
&= \frac{\theta_1^{z_1} (1\theta_1)^{N_1 z_1} \theta_1^{z_2} (1\theta_2)^{N_2 z_2} \theta_1^{a_1 1} (1\theta_1)^{b_1 - 1} \theta_2^{a_2 1} (1\theta_2)^{b_2 - 1}}{p(D) B(a_1, b_1) B(a_2, b_2)} \\
&= \frac{\theta_1^{z_1 + a_1 1} (1\theta_1)^{N_1 z_1 + b_1 1} \theta_2^{z_2 + a_2 1} (1\theta_2)^{N_2 z_2 + b_2 1}}{p(D) B(a_1, b_1) B(a_2, b_2)}
\end{aligned}$$

So the posterior distribution of $\langle \theta_1, \theta_2 \rangle$ is two independent Beta distributions: $\mathsf{Beta}(z_1 + a, N_1 - z_1 + b_1)$ and $\mathsf{Beta}(z_2 + a, N_2 - z_2 + b_2)$.

Some nice images of these distributions appear on page 167.



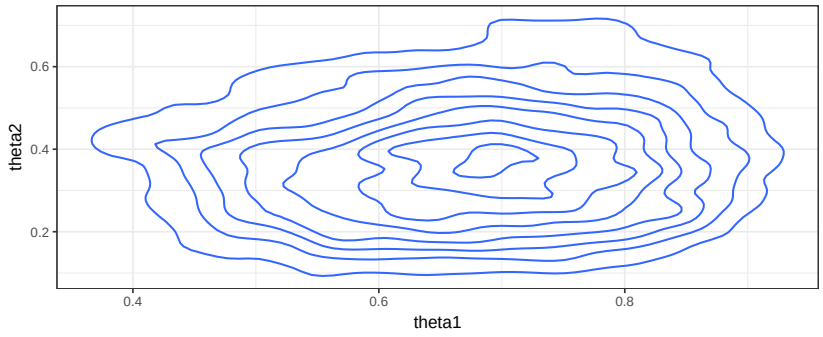
7.6.3 Metropolis

```
metro_2coins <- function(
  z1, n1,              # z = successes, n = trials
  z2, n2,              # z = successes, n = trials
  size  = c(0.1, 0.1), # sds of jump distribution
  start = c(0.5, 0.5), # value of thetas to start at
  num_steps = 5e4,     # number of steps to run the algorithm
  prior1 = dbeta,      # function describing prior
  prior2 = dbeta,      # function describing prior
  args1 = list(),      # additional args for prior1
  args2 = list()      # additional args for prior2
  ) {

  theta1           <- rep(NA, num_steps)  # trick to pre-alocate memory
  theta2           <- rep(NA, num_steps)  # trick to pre-alocate memory
  proposed_theta1  <- rep(NA, num_steps)  # trick to pre-alocate memory
  proposed_theta2  <- rep(NA, num_steps)  # trick to pre-alocate memory
  move             <- rep(NA, num_steps)  # trick to pre-alocate memory
  theta1[1]        <- start[1]
```

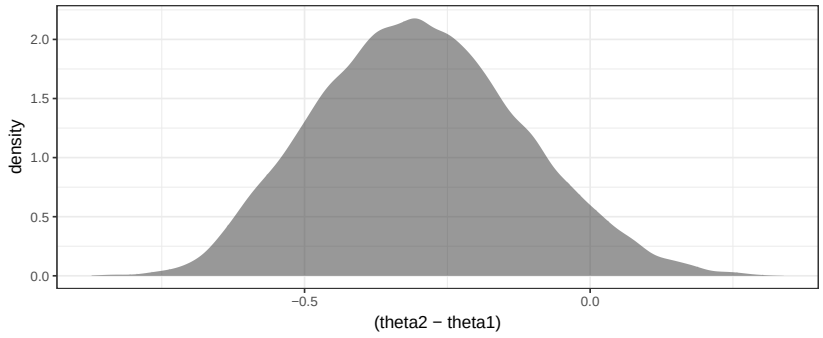
```
theta2[1]         <- start[2]

size1 <- size[1]
size2 <- size[2]

for (i in 1:(num_steps-1)) {
  # head to new "island"
  proposed_theta1[i + 1] <- rnorm(1, theta1[i], size1)
  proposed_theta2[i + 1] <- rnorm(1, theta2[i], size2)

  if (proposed_theta1[i + 1] <= 0 ||
      proposed_theta1[i + 1] >= 1 ||
      proposed_theta2[i + 1] <= 0 ||
      proposed_theta2[i + 1] >= 1) {
    proposed_posterior <- 0  # because prior is 0
  } else {
    current_prior <-
      do.call(prior1, c(list(theta1[i]), args1)) *
      do.call(prior2, c(list(theta2[i]), args2))
    current_likelihood  <-
      dbinom(z1, n1, theta1[i]) *
      dbinom(z2, n2, theta2[i])
    current_posterior   <- current_prior * current_likelihood

    proposed_prior <-
      do.call(prior1, c(list(proposed_theta1[i+1]), args1)) *
      do.call(prior2, c(list(proposed_theta2[i+1]), args2))
    proposed_likelihood  <-
      dbinom(z1, n1, proposed_theta1[i+1]) *
      dbinom(z2, n2, proposed_theta2[i+1])
    proposed_posterior   <- proposed_prior * proposed_likelihood
  }
  prob_move           <- proposed_posterior / current_posterior

  # sometimes we "sail back"
  if (runif(1) > prob_move) { # sail back
     move[i + 1] <- FALSE
    theta1[i + 1] <- theta1[i]
    theta2[i + 1] <- theta2[i]
  } else {                    # stay
     move[i + 1] <- TRUE
    theta1[i + 1] <- proposed_theta1[i + 1]
    theta2[i + 1] <- proposed_theta2[i + 1]
  }
}

tibble(
  step = 1:num_steps,
  theta1 = theta1,
  theta2 = theta2,
  proposed_theta1 = proposed_theta1,
  proposed_theta2 = proposed_theta2,
  move = move,
```

```
    size1 = size1,
    size2 = size2
  )
}
```

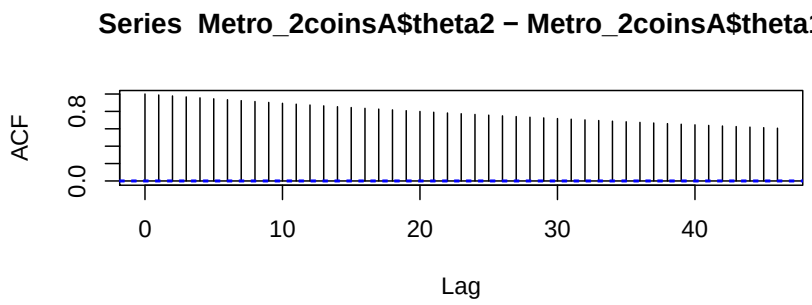
```
Metro_2coinsA <-
  metro_2coins(
    z1 = 6, n1 = 8,
    z2 = 2, n2 = 7,
    size = c(0.02, 0.02),
    args1 = list(shape1 = 2, shape2 = 2), args2 = list(shape1 = 2, shape2 = 2)
  )

Metro_2coinsA %>%
  gf_density2d(theta2 ~ theta1)
```




```
Metro_2coinsA %>%
  gf_density(~ (theta2 - theta1))
```

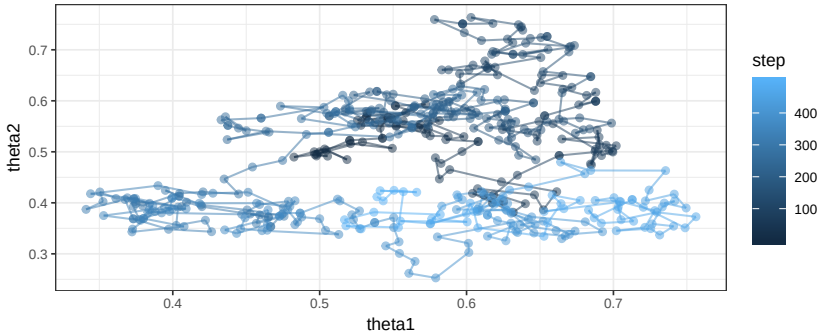



```
# effective sample size is much smaller than apparent sample size due to auto-correlation
acf(Metro_2coinsA$theta2 - Metro_2coinsA$theta1)
```

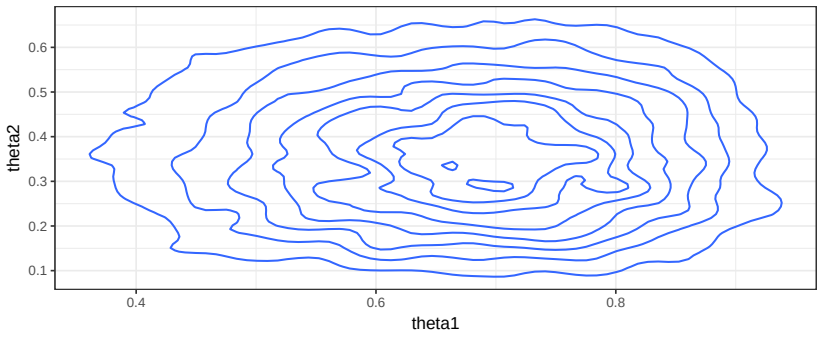



```
Metro_2coinsA %>% filter(step < 500) %>%
```

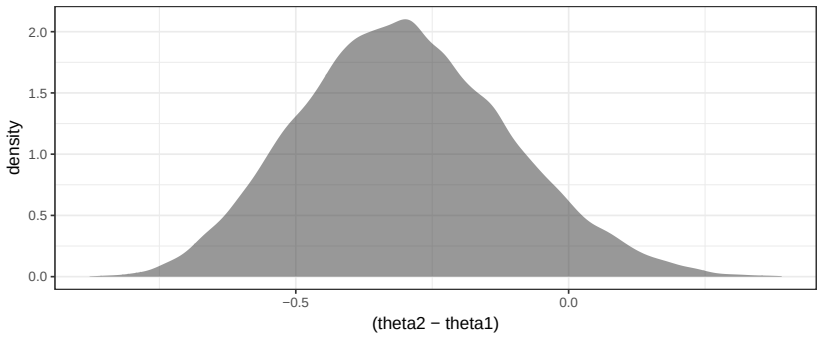
```
  gf_path(theta2 ~ theta1, color = ~ step, alpha = 0.5) %>%
  gf_point(theta2 ~ theta1, color = ~ step, alpha = 0.5)
```



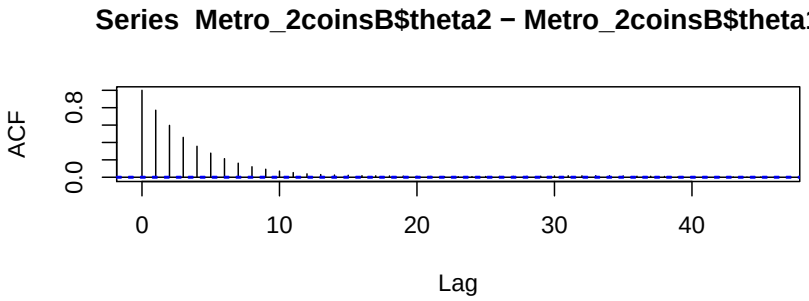

```
Metro_2coinsB <-
  metro_2coins(
    z1 = 6, n1 = 8,
    z2 = 2, n2 = 7,
    size = c(0.2, 0.2),
    args1 = list(shape1 = 2, shape2 = 2), args2 = list(shape1 = 2, shape2 = 2)
  )

Metro_2coinsB %>%
  gf_density2d(theta2 ~ theta1)
```



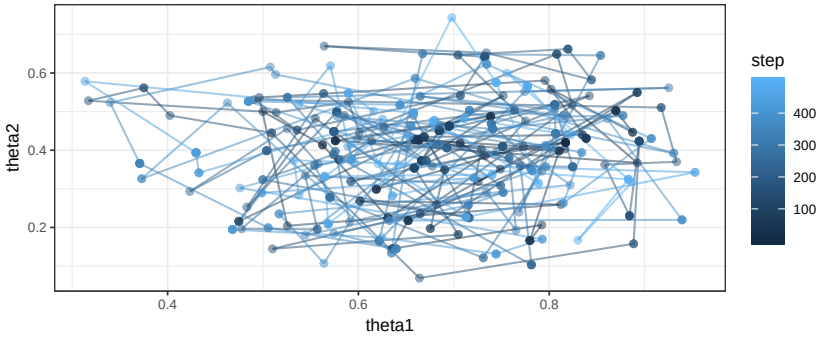

```
Metro_2coinsB %>%
  gf_density(~ (theta2 - theta1))
```




```
# effective sample size is better but still quite a bit
# smaller than apparent sample size due to auto-correlation
acf(Metro_2coinsB$theta2 - Metro_2coinsB$theta1)
```



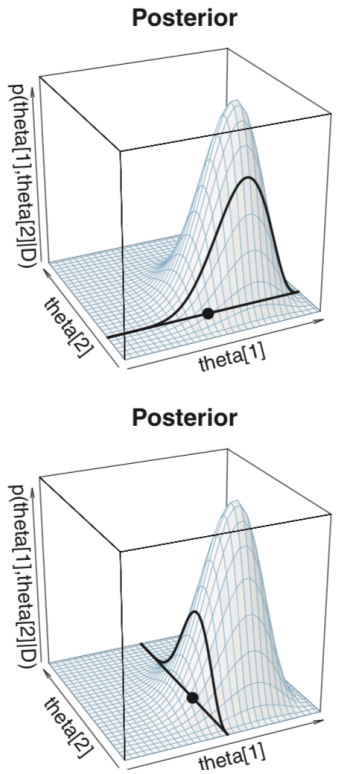

```
Metro_2coinsB %>% filter(step < 500) %>%
  gf_path(theta2 ~ theta1, color = ~ step, alpha = 0.5) %>%
  gf_point(theta2 ~ theta1, color = ~ step, alpha = 0.5)
```



7.6.4 Gibbs sampling

Gibbs sampling provides an attempt to improve on the efficiency of the standard Metropolis algorithm by using a different method to propose new parameter values. The idea is this:

- Pick one of the parameter values: θ_i
- Determine the posterior distribution of θ_i using current estimates of the other parameters $\{\theta_j \mid j \neq i\}$
  - This won't be exactly right, because those estimates are not exactly right, but it should be good when the parameters estimates are close to correct.
- Sample from the posterior distribution for θ_i to get a new proposed value for θ_i.
  - Always accept this proposal, since it is being sampled from a distribution that already makes more likely values more likely to be proposed.
- Keep cycling through all the parameters, each time updating one using the current estimates for the others.



It takes some work to show that this also converges, and in practice it is often more efficient than the basic Metropolis algorithm. But it is limited to situations where the marginal posterior can be be sampled from.

In our example, we can work out the marginal posterior fairly easily:

$$
\begin{aligned}
p(\theta_1|\theta_2, D) &= p(\theta_1, \theta_2|D)/p(\theta_2|D) \\
&= p(\theta_1, \theta_2|D) \int d\theta_1 p(\theta_1, \theta_2|D) \\
&= \frac{\text{dbeta}(\theta_1, z_1 + a_1, N_1 z_1 + b_1) \cdot \text{dbeta}(\theta_2, z_2 + a_2, N_2 z_2 + b_2)}{\int d\theta_1 \text{dbeta}(\theta_1, z_1 + a_1, N_1 z_1 + b_1) \cdot \text{dbeta}(\theta_2 \mid z_2 + a_2, N_2 z_2 + b_2)} \\
&= \frac{\text{dbeta}(\theta_1, z_1 + a_1, N_1 z_1 + b_1) \cdot \text{dbeta}(\theta_2, z_2 + a_2, N_2 z_2 + b_2)}{\text{dbeta}(\theta_2 \mid z_2 + a_2, N_2 z_2 + b_2) \int d\theta_1 \text{dbeta}(\theta_1, z_1 + a_1, N_1 z_1 + b_1)} \\
&= \frac{\text{dbeta}(\theta_1, z_1 + a_1, N_1 z_1 + b_1)}{\int d\theta_1 \text{dbeta}(\theta_1, z_1 + a_1, N_1 z_1 + b_1)} \\
&= \text{dbeta}(\theta_1, z_1 + a_1, N_1 z_1 + b_1)
\end{aligned}
$$

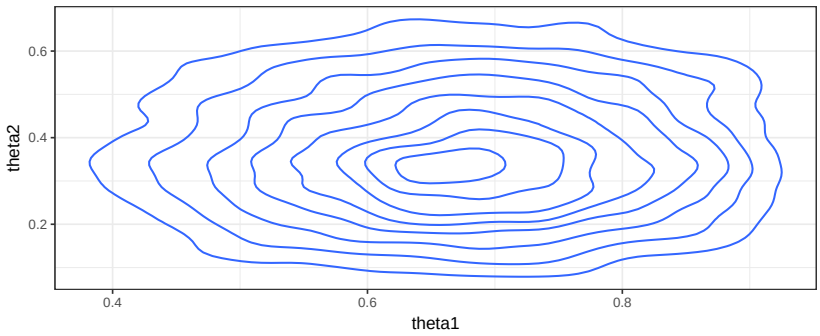
In other words, $p(\theta_1 \mid \theta_2, D) = p(\theta_1, D)$, which isn't surprising since we already know that the posterior distributions θ_1 and θ_2 are independent. (In more complicated models, the marginal posterior may depend on all of the parameters, so the calculation above might not be so simple.) Of course, the analogous statement holds for θ_2 in this model.

This examples shows off Gibbs sampling in a situation where it shines: when the posterior distribution is a collection of independent marginals.

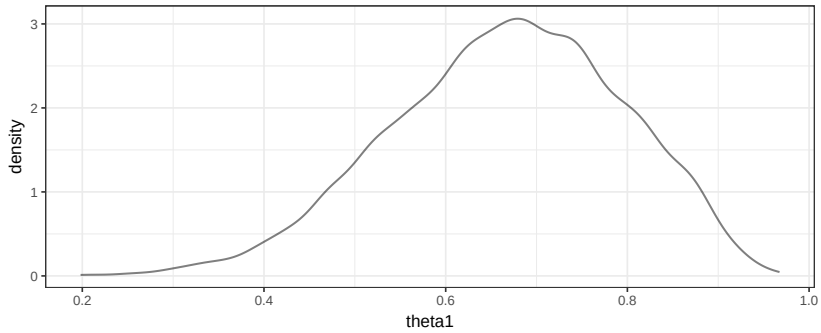
```
gibbs_2coins <- function(
  z1, n1,              # z = successes, n = trials
  z2, n2,              # z = successes, n = trials
  start = c(0.5, 0.5), # value of thetas to start at
```

```
  num_steps = 1e4,      # number of steps to run the algorithm
  a1, b1,               # params for prior for theta1
  a2, b2                # params for prior for theta2
  ) {

  theta1            <- rep(NA, num_steps)  # trick to pre-alocate memory
  theta2            <- rep(NA, num_steps)  # trick to pre-alocate memory
  theta1[1]         <- start[1]
  theta2[1]         <- start[2]

  for (i in 1:(num_steps-1)) {
    if (i %% 2 == 1) { # update theta1
      theta1[i+1] <- rbeta(1, z1 + a1, n1 - z1 + b1)
      theta2[i+1] <- theta2[i]
    } else {            # update theta2
      theta1[i+1] <- theta1[i]
      theta2[i+1] <- rbeta(1, z2 + a2, n2 - z2 + b2)
    }
  }

  tibble(
    step = 1:num_steps,
    theta1 = theta1,
    theta2 = theta2,
  )
}
```

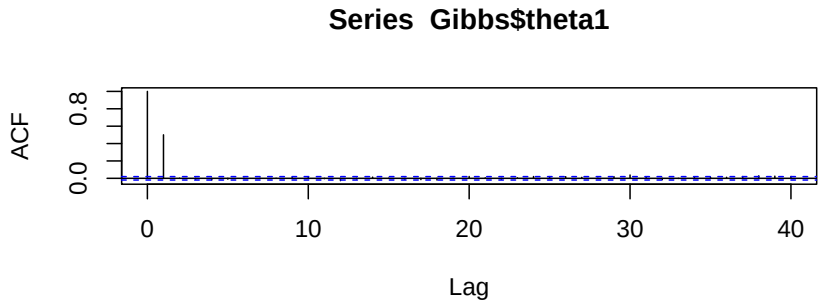
```
Gibbs <-
  gibbs_2coins(z1 = 6, n1 = 8,
               z2 = 2, n2 = 7,
               a1 = 2, b1 = 2, a2 = 2, b2 = 2)

Gibbs %>% gf_density2d(theta2 ~ theta1)
```



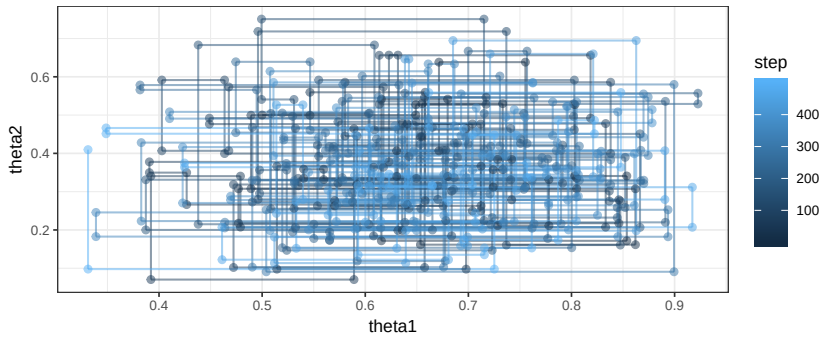

```
Gibbs %>% gf_dens( ~ theta1)
```




```
acf(Gibbs$theta1)
```

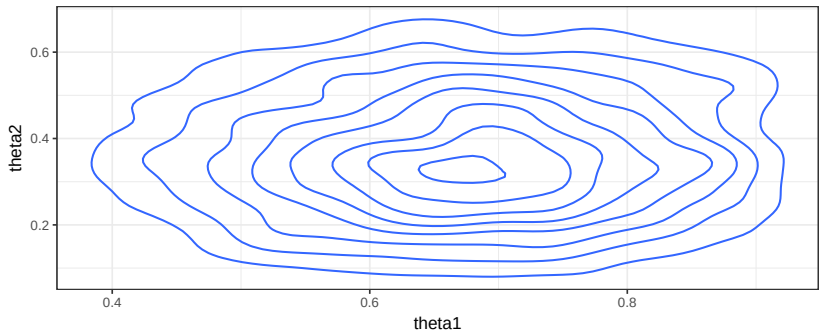



```
Gibbs %>% filter(step < 500) %>%
  gf_path(theta2 ~ theta1, color = ~ step, alpha = 0.5) %>%
  gf_point(theta2 ~ theta1, color = ~ step, alpha = 0.5)
```

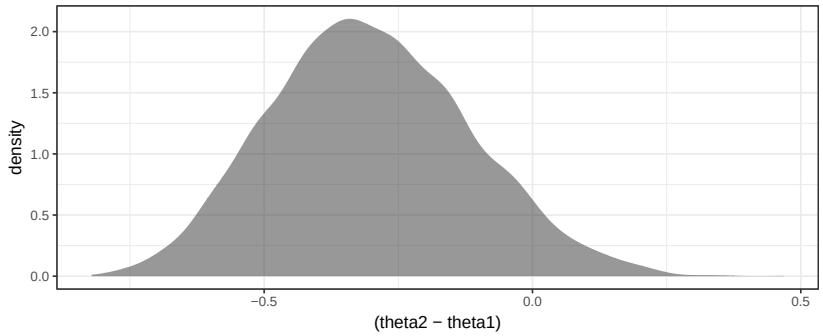


Note: Software like JAGS only records results each time it makes a complete cycle through the parameters. We could do that by keeping every other row:

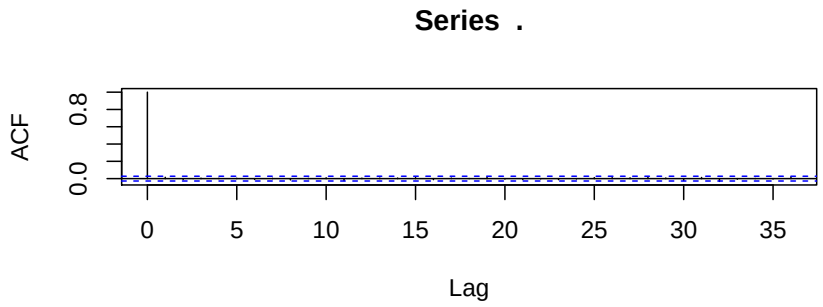
```
Gibbs %>% filter(step %% 2 == 0) %>% gf_density2d(theta2 ~ theta1)
```



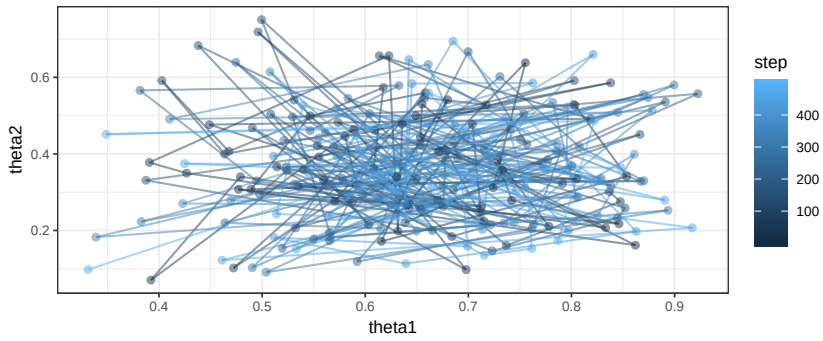

```
Gibbs %>% filter(step %% 2 == 0) %>% gf_density( ~ (theta2 - theta1))
```




```
Gibbs %>% filter(step %% 2 == 0) %>%
  mutate(difference = theta2 - theta1) %>%
  pull(difference) %>%
  acf()
```




```
Gibbs %>% filter(step < 500, step %% 2 == 0) %>%
  gf_path(theta2 ~ theta1, color = ~ step, alpha = 0.5) %>%
  gf_point(theta2 ~ theta1, color = ~ step, alpha = 0.5)
```



7.6.5 Advantages and Disadvantages of Gibbs vs Metropolis

Advantages

- **No need to tune**. The preformance of the Metropolis algorithm depends on the jump rule used. Gibbs "auto-tunes" by using the marginal posteriors.
- Gibbs sampling has the **potential to sample much more efficiently** than Metropolis. It doesn't propose updates only to reject them, and doesn't suffer from poor tuning that can make Metropolis search very slowly through the posterior space.

Disadvantages

- Less general: **We need to be able to sample from the marginal posterior**.
- Gibbs sampling **can be slow if the posterior has highly correlated parameters** since given all but one of them, the algorithm with be very certain about the remaining one and adjust it only a very small amount. It is hard for Gibbs samplers to quickly move along "diagonal ridges" in the posterior.

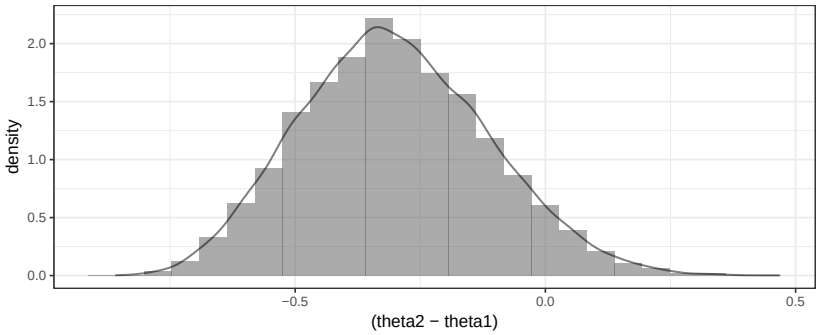
The good news is that other people have done the hard work of coding up Gibbs sampling for us, we just need to learn how to describe the model in a way that works with their code. We will be using JAGS (just another Gibbs Sampler) and later Stan (which generalizes the metropolis algorithm in a different way) to fit more interesting models that would be too time consuming to custom code on our own. The examples in the chapter are to help us understand better how these algorithms work so we can interpret the results we obtian when using them.

7.6.6 So what do we learn about the coins?

We have seen that we can fit this model a number of different ways now, but we haven't actually looked at the results. Are the coins different? How much different?

Since the Gibbs sampler is more efficient than the Metropolis algorithm for this situation, we'll use the posterior samples from the Gibbs sampler to answer. We are interested in $\theta_2 - \theta_1$, the difference in the biases of the two coins. To learn about that, we simply investigate the distribution of that quantity in our posterior samples. (Note: You cannot learn about this difference by only considering θ_1 and θ_2 sepearately.)

```
gf_dhistogram(~(theta2 - theta1), data = Gibbs) %>%
  gf_dens()
```




```
hdi(Gibbs %>% mutate(difference = theta2 - theta1), pars = "difference")
```

par	lo	hi	prob
difference	-0.6623	0.057	0.95

```
mosaic::prop(~(theta2 - theta1 > 0), data = Gibbs)
```

```
## prop_TRUE
##    0.0601
```

We don't have much data, so the posterior distribution of the difference in biases is pretty wide. 0 is within the 95% HDI for $\theta_2 - \theta_1$, so we don't have compelling evidence that the two biases are different based on this small data set.

7.7 MCMC posterior sampling: Big picture

7.7.1 MCMC = Markov chain Monte Carlo

- Markov chain because the process is a Markov process with probabilities of transitioning from one state to another
- Monte Carlo because it inovles randomness. (Monte Carlo is a famous casino location.)

We will refer to one random walk starting from a pariticular starting value as a **chain**.

7.7.2 Posterior sampling: Random walk through the posterior

Our goal, whether we use Metropolis, Gibbs sampling, Stan, or some other MCMC posterior samping method, is to generate random values from the posterior distribution. Ideally these samples should be

- **representative** of the posterior and not of other artifacts like the starting location of the chain, or tuning parameters used to generate the walk.
- **accurate** to the posterior. If we run the algorithm multiple times, we would like the results to be similar from run to run. (They won't match exactly, but if they are all giving good approximations to the same thing, then they should all close to each other.)
- **efficient**. The theory says that all of these methods converge to the correct posterior distribution, but in practice, we can only do a finite run. If the sampling is not efficient enough, our finite run might give us a distorted picture (that would eventually have been corrected had we run things long enough).

If we are convinced that the posterior sampling is of high enough quality, we can use the posterior samples to answer all sorts of questions relatively easily.

7.7.3 Where do we go from here?

1. Learn to design more interesting models to answer more interesting questions.
2. Learn to describe these models and hand them to JAGS or Stan for posterior sampling.
3. Learn to diagnose posterior samples to detect potential problems with the posterior sampling.
4. Learn how to interpret the results.

7.8 Exercises

1. In this exercise, you will see how the Metropolis algorithm operates with a multimodal prior.
   a. Define the function $p(\theta) = (cos(4\pi\theta) + 1)^2/1.5$ in R.
   b. Use `gf_function()` to plot $p(\theta)$ on the interval from 0 to 1. [Hint: Use the `xlim` argument.]
   c. Use `integrate()` to confirm that p is a pdf.
   d. Run `metro_bern()` with p as your prior, with no data (`x = 0`, `n = 0`), and with `size = 0.2`. Plot the posterior distribution of θ and explain why it looks the way it does.
   e. Now create a posterior histogram or density plot using `x = 2`, `n = 3`. Do the results look reasonable? Explain.
   f. Now create a posterior histogram or density plot with `x = 1`, `n = 3`, and `size = 0.02`. Comment on how this compares to plot you made in the previous item.

g. Repeat the previous two items but with `start = 0.15` and `start = 0.95`. How does this help explain what is happening? Why is it good practice to run MCMC algorithms with several different starting values as part of the diagnositc process?

h. How would looking at trace plots from multiple starting points help you detect this problem? (What would the trace plots look like when things are good? What would they look like when things are bad?)

Chapter 8

JAGS – Just Another Gibbs Sampler

This chapter focuses on a very simple model – one for which JAGS is overkill. This allows us to get familiar with JAGS and the various tools to investigate JAGS models in a simple setting before moving on to more interesting models soon.

8.1 What JAGS is

JAGS (Just Another Gibbs Sampler) is an implementation of an MCMC algorithm called Gibbs sampling to sample the posterior distribution of a Bayesian model.

We will interact with JAGS from within R using the following packages:

- R2jags – interface between R and JAGS
- coda – general tools for analyzing and graphing MCMC algorithms
- bayesplot – a number of useful plots using `ggplot2`
- CalvinBayes – includes some of the functions from Kruschke's text and other things to make our lives better.

8.1.1 JAGS documentation

You can find JAGS documentation at http://people.stat.sc.edu/hansont/stat740/jags_user_manual.pdf. This can be useful if you need to find out particulars about things like the distributions that are availble in JAGS.

8.1.2 Updating C and CLANG

Based on limited testing, it appears that things are good to go and you should not need to do this.

~~To use the newest versions of JAGS, Stan, and the R packages that accompany them, we need to use a newer version of some software than is standard for <rstudio.calvin.edu>. I have taken care of this at the system level, and that may suffice, but if things don't work in your account, take the following steps:~~

1. ~~Open a terminal with Tools > Terminal > New Terminal~~
2. ~~Copy-and-paste this into the terminal window.~~

```
echo "source scl_source enable devtoolset-7 llvm-toolset-7" >> ~/.bashrc
```

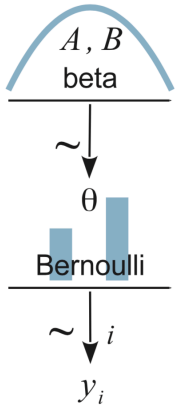
~~This tells the server to use a newer version of C++ and CLANG.~~

3. ~~Close the terminal~~
4. ~~Restart R with Session > Restart R~~

~~**You should only need to go through these steps once.**~~

8.2 Example 1: estimating a proportion

8.2.1 The Model



That is

$$
\begin{aligned}
Y_i &\sim \mathsf{Bern}(\theta) \\
\theta &\sim \mathsf{Beta}(a, b)
\end{aligned}
$$

8.2.2 Load Data

The data sets provided as csv files by Kruschke also live in the `CalvinBayes` package, so you can read this file with

```
library(CalvinBayes)
data("z15N50")
glimpse(z15N50)
```

```
## Observations: 50
## Variables: 1
## $ y <dbl> 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1, ...
```

We see that the data are coded as 50 0's and 1's in a variable named `y`. (You should use better names when creating your own data sets.)

8.2.3 Specify the model

There are at least three R packages that provide an interface to JAGS: rjags, R2jags, and runjags. We will primarily use R2jags. Kruschke primarily uses rjags. The main advantage of R2jags is that we can specify the model by creating a special kind of function. [1] The avoids the need to create temporary files (as rjags requires) and keeps things tidier in our R markdown documents.

[1] This is a bit of a trick that R2jags uses. The function created is never run. The code is inspected and taken as the description of the model. If you were to run the funtion, all it would do is create R formulas.

The main part of the model description is the same in either style, but notice that the using the function style, we do not need to include `model{ ... }` in our description. Here's how we describe our simple model.

```
bern_model <- function() {
  for (i in 1:N) {
    y[i] ~ dbern(theta)  # each response is Bernoulli with fixed parameter theta
  }
  theta ~ dbeta(1, 1)    # prior for theta
}
```

Some things to note:

- `dbern()` and `dbeta()` are JAGS functions. The JAGS distribution functions are similar to, but not identical to the ones in R. (R doesn't have Bernoulli at all, for example.) Sometimes the parameterization are different. Importantly, JAGS doesn't have named arguments, so the arguments must go in the order JAGS requires. Notice that we are only giving the distribution name and its parameters. (So the first argument that R requires is not part of this in JAGS.)
- JAGS is also not vectorized the way R is, so we will need to write some explicit for loops to say "do this to every that". In the example above, the for loops says that for each row of the data (`i in 1:N`), the response (`y[i]`) is Bernoulli with paramter θ (`dbern(theta)`).

8.2.4 Run the model

`R2jags::jags()` can be used to run our JAGS model. We need to specify three things: (1) the model we are using (as defined above), (2) the data we are using, (3) the parameters we want saved in the posterior sampling. (`theta` is the only parameter in this model, but in larger models, we might choose to save only some of the parameters).

The data do not need to be in a data frame, and this usually means a bit more work on our part to tell JAGS things like how much data there is. We will prepare all the information JAGS needs about the data in a **list** using `list()`.

There are some additional, optional things we might want to control as well.
More on those later. For now, let's fit the model using the default values for everything else.

```
# Load the R2jags package
library(R2jags)

# Make the same "random" choices each time this is run.
# This makes the Rmd file stable so you can comment on specific results.
set.seed(123)

# Fit the model
bern_jags <-
  jags(
    data = list(y = z15N50$y, N = nrow(z15N50)),
    model.file = bern_model,
    parameters.to.save = c("theta")
  )
```

```
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 50
##    Unobserved stochastic nodes: 1
```

```
##    Total graph size: 53
##
## Initializing model
```

Let's take a quick look at what we have.

```
bern_jags
```

```
## Inference for Bugs model at "/var/folders/py/txwd26jx5rq83f4nn0f5fmmm0000gn/T//RtmpkAYhBH/model1121db6
##  3 chains, each with 2000 iterations (first 1000 discarded)
##  n.sims = 3000 iterations saved
##          mu.vect sd.vect   2.5%    25%    50%    75%  97.5%  Rhat n.eff
## theta      0.308   0.064  0.191  0.261  0.306  0.351  0.435 1.001  3000
## deviance  62.089   1.395 61.087 61.186 61.571 62.459 65.770 1.001  3000
##
## For each parameter, n.eff is a crude measure of effective sample size,
## and Rhat is the potential scale reduction factor (at convergence, Rhat=1).
##
## DIC info (using the rule, pD = var(deviance)/2)
## pD = 1.0 and DIC = 63.1
## DIC is an estimate of expected predictive error (lower deviance is better).
```

Some notes on the output above:

- `3 chains`: The Gibbs sampler was run 3 times with 3 different starting values. Each chain ran for 2000 steps, but only the last 1000 steps were saved.
- `n.sims = 3000` (1000 in each of 3 chains).
- `mu.vect`: We see that the average value of `theta` in our posterior sample is 0.308.

- `n.eff = 3000` is the number of effective samples. In this case, JAGS is being very efficient, as we would expect since it is just sampling directly from the posterior distribution.
- `Rhat = 1`: This is a check for possible convergence problems. If an MCMC sampler has converged, `Rhat` will be 1. So if the value we see is not very close to 1, that is a sign of problems. Any value greater than 1.1 is a cause for concern.
- We'll talk more about deviance later.

8.3 Extracting information from a JAGS run

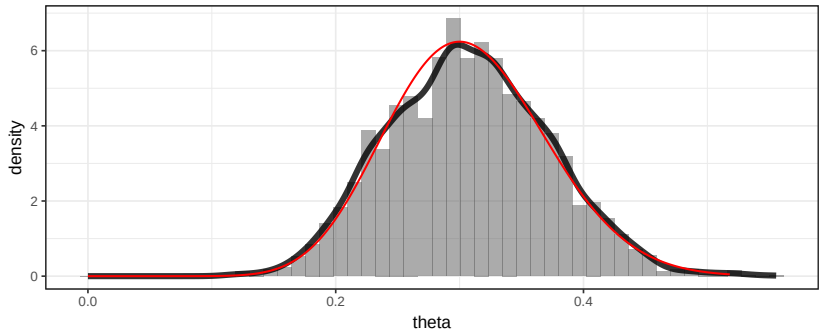
8.3.1 posterior()

We can plot the posterior distribution, using `posterior()` to extract the posterior samples as a data frame. Since we know the posterior distribution should be Beta(16, 36), we'll add that to our plot as a reference to see how well our posterior sample is doing.

```
library(CalvinBayes)
head(posterior(bern_jags))
```

deviance	theta	chain	iter
61.65	0.2528	chain:1	1
61.11	0.2895	chain:1	2
62.74	0.3871	chain:1	3
61.29	0.3296	chain:1	4
61.09	0.3052	chain:1	5
63.67	0.4098	chain:1	6

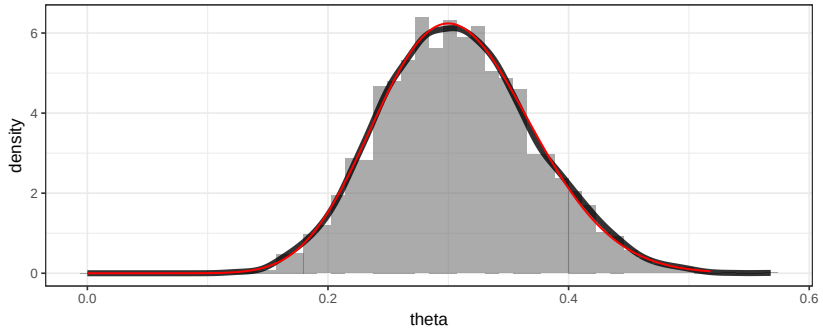
```
gf_dhistogram(~theta, data = posterior(bern_jags), bins = 50) %>%
  gf_dens(~theta, size = 1.5, alpha = 0.8) %>%
  gf_dist("beta", shape1 = 16, shape2 = 36, color = "red")
```



8.3.2 Side note: posterior sampling and the grid method

It is also possible to generate posterior samples when we use the grid method. The mosaic package include the `resample()` function that will sample rows of a data frame with replacement using specified probabilities (given by the posterior, for example). Here's how that works.

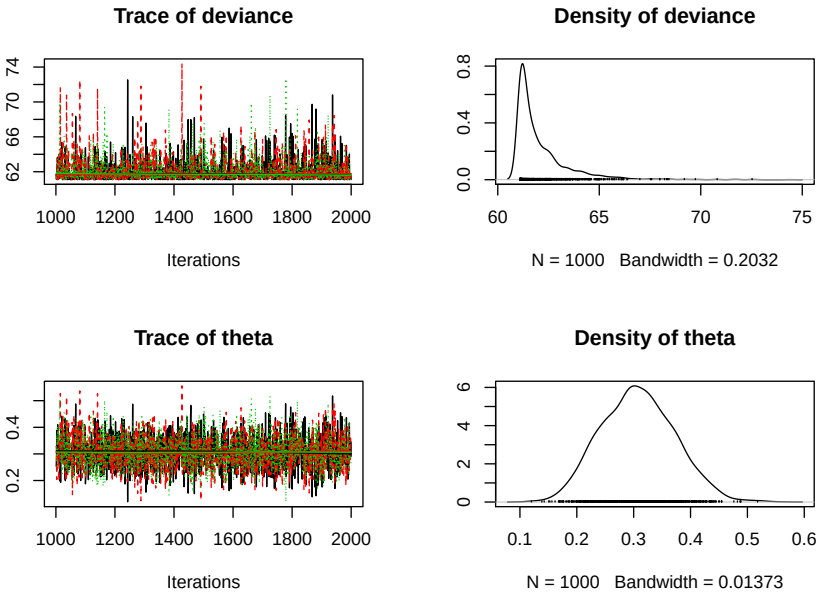
```
Grid <-
  expand.grid(
    theta = seq(0, 1, by = 0.001)
  ) %>%
  mutate(
    prior = dbeta(theta, 1, 1),
    likelihood = dbinom(15, 50, theta),
    posterior = prior * likelihood,
    posterior = posterior / sum(posterior) / 0.001
  )

Posterior <- resample(Grid, size = 5000, prob = Grid$posterior)
gf_dhistogram(~ theta, data = Posterior, bins = 50) %>%
  gf_dens(~theta, size = 1.5, alpha = 0.8) %>%
  gf_dist("beta", shape1 = 16, shape2 = 36, color = "red")
```



8.3.3 Using coda

The `coda` package provides output analysis and diagnostics for MCMC algorithms. In order to use it, we must convert our JAGS object into something `coda` recognizes. We do with with the `as.mcmc()` function.

```
bern_mcmc <- as.mcmc(bern_jags)
plot(bern_mcmc)
```

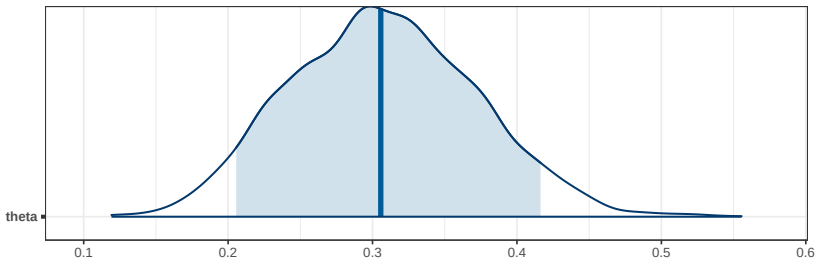


Note: Kruschke uses `rjags` without `R2jags`, so he does this step using `rjags::coda.samples()` instead of `as.mcmc()`. Both functions result in the same thing – posterior samples in a format that `coda` expects, but they have different starting points.

8.3.4 Using bayesplot

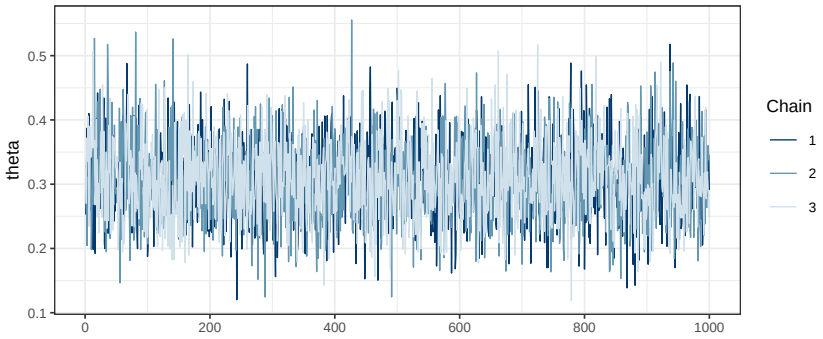
The mcmc object we extracted with `as.mcmc()` can be used by the utilities in the `bayesplot()`. Here, for example is the `bayesplot` plot of the posterior distribution for theta. By default, a vertical line segment is drawn at the median of the posterior distribution.

```
library(bayesplot)
mcmc_areas(
  bern_mcmc,
  pars = c("theta"),      # make a plot for the theta parameter
  prob = 0.90)            # shade the central 90%
```



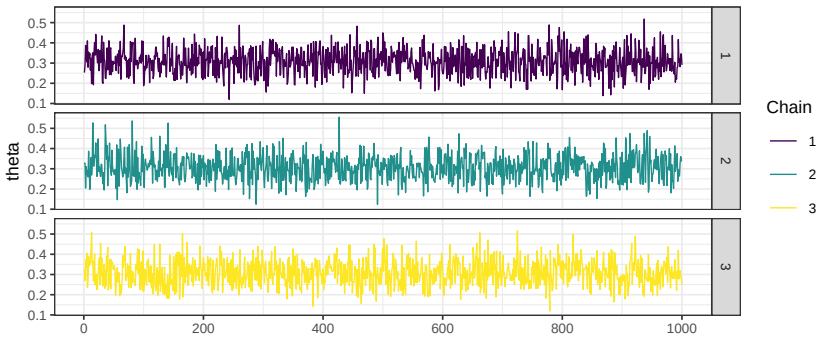
One advantage of `bayesplot` is that the plots use the `ggplot2` system and so interoperate well with `ggformula`.

```
mcmc_trace(bern_mcmc, pars = "theta")
```




```
# spearate the chains using facets and modify the color scheme
mcmc_trace(bern_mcmc, pars = "theta") %>%
  gf_facet_grid(Chain ~ .) %>%
  gf_refine(scale_color_viridis_d())
```

```
## Scale for 'colour' is already present. Adding another scale for
## 'colour', which will replace the existing scale.
```



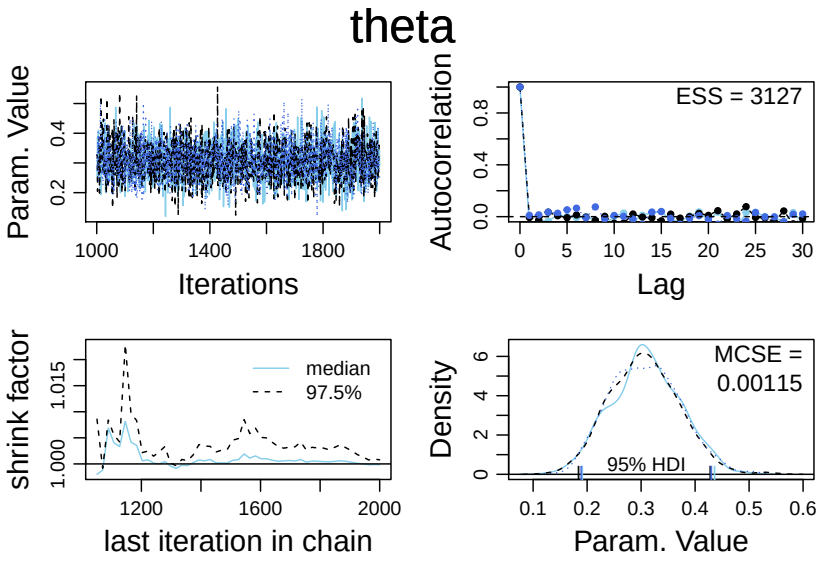
We will encounter additional plots from `bayesplot` as we go along.

8.3.5 Using Kruschke's functions

I have put (modified versions of) some of functions from Kruschke's book into the `CalvinBayes` package so that you don't have to source his files to use them.

diag_mcmc() [diagMCMC()]

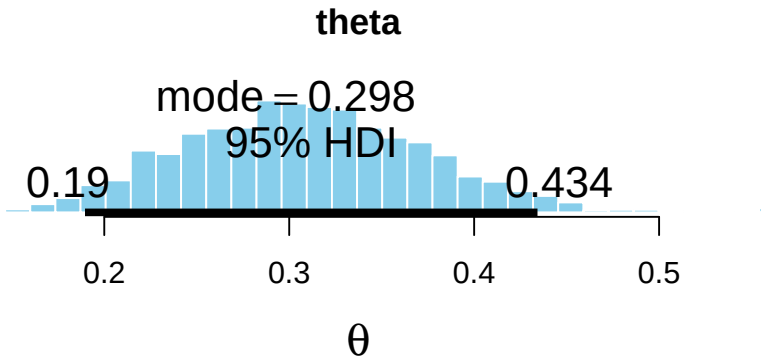
```
diag_mcmc(bern_mcmc, par = "theta")
```



plot_post() [plotPost()]

The `plot_post()` function takes as its first argument a vector of posterior sampled values for one of the parameters. We can extract such a vector a couple different ways:

```
## # these produce the same output
plot_post(bern_mcmc[, "theta"], main = "theta", xlab = expression(theta))
```

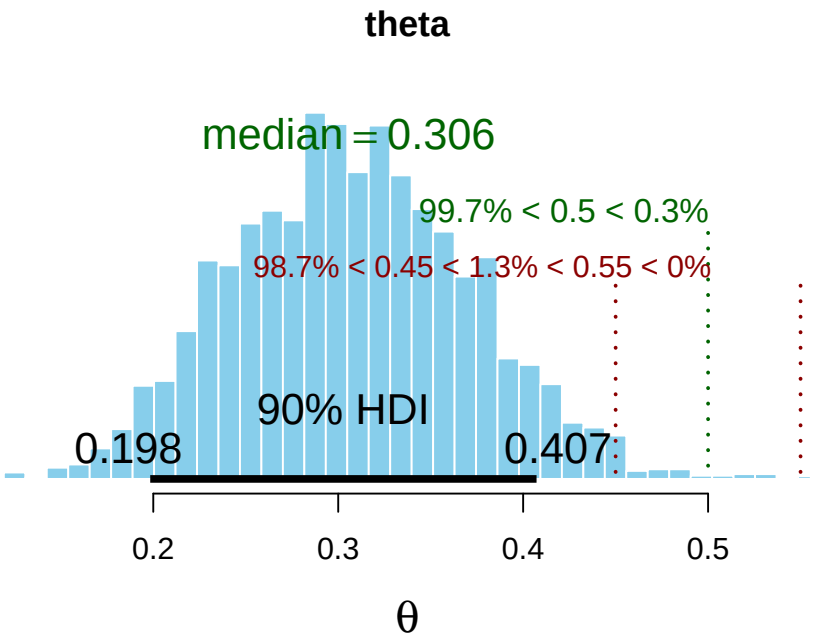



```
## $posterior
##       ESS   mean median   mode
## var1 3000 0.3075 0.3056 0.2982
##
## $hdi
##   prob     lo     hi
## 1 0.95 0.1896 0.4342
```

```
## plot_post(posterior(bern_jags)$theta, main = "theta", xlab = expression(theta))
```

There are a number of options that allow you to add some additional information to the plot. Specifying `quietly = TRUE` will turn off the numerical display that `plot_post()` generates along with the plot. Here is an example.[^08-2]

```
plot_post(bern_mcmc[, "theta"], main = "theta", xlab = expression(theta),
          cenTend = "median", compVal = 0.5, ROPE = c(0.45, 0.55),
          credMass = 0.90, quietly = TRUE)
```



8.4 Optional arguments to jags()

8.4.1 Number and size of chains

Sometimes we want to use more or longer chains (or fewer or shorter chains if we are doing a quick preliminary check before running longer chains later). `jags()` has three arguments for this:

- `n.chains`: number of chains
- `n.iter`: number of iterations per chain
- `n.burnin`: number of burn in steps per chain
- `n.thin`: keep one sample per `n.thin`.

The default value of `n.thin` is set to save about 1000 values per chain. So in the example below, we end up with only 4000 samples (1000 per chain) rather than the 16000 you might have expected.

```
set.seed(76543)
bern_jags2 <-
  jags(
    data = list(y = z15N50$y, N = nrow(z15N50)),
    model.file = bern_model,
    parameters.to.save = c("theta"),
    n.chains = 4, n.iter = 5000, n.burnin = 1000,
  )
bern_jags2
```

Setting `n.thin = 1` will save them all.

```
set.seed(76543)
bern_jags2a <-
  jags(
    data = list(y = z15N50$y, N = nrow(z15N50)),
    model.file = bern_model,
    parameters.to.save = c("theta"),
    n.chains = 4, n.iter = 5000, n.burnin = 1000,
```

```
    n.thin = 1
  )
bern_jags2a
```

8.4.2 Starting point for chains

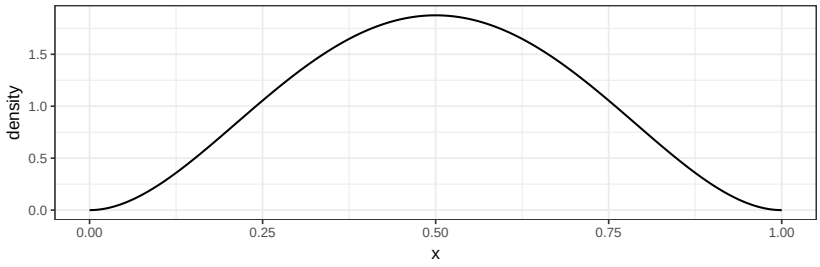
We can also control the starting point for the chains. Starting different chains and quite different parameter values can help

- verify that the MCMC algorithm is not overly sensitive to where we are starting from, and
- ensure that the MCMC algorithm has explored the posterior distribution sufficiently.

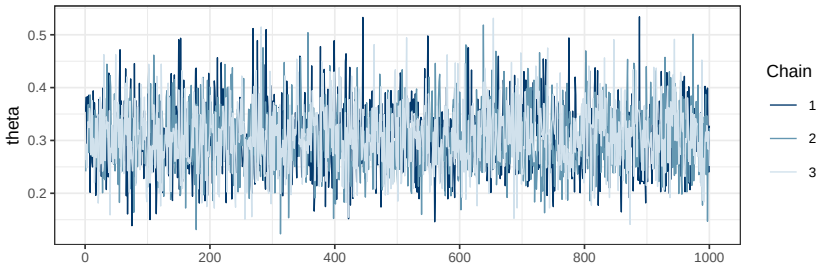
On the other hand, if we start a chain too far from the peak of the posterior distribution, the chain may have trouble converging.

We can provide either specific starting points for each chain or a function that generates random starting points.

```
gf_dist("beta", shape1 = 3, shape2 = 3)
```




```
set.seed(2345)
bern_jags3 <-
  jags(
    data = list(y = z15N50$y, N = nrow(z15N50)),
    model.file = bern_model,
    parameters.to.save = c("theta"),
    # start each chain by sampling from the prior
    inits = function() list(theta = rbeta(1, 3, 3))
  )

bern_jags4 <-
  jags(
    data = list(y = z15N50$y, N = nrow(z15N50)),
    model.file = bern_model,
    parameters.to.save = c("theta"),
    # choose specific starting point for each chain
    inits = list(
      list(theta = 0.5), list(theta = 0.7), list(theta = 0.9)
    )
  )

mcmc_trace(as.mcmc(bern_jags4), pars = "theta")
```



It is a good sign that our three traces look very similar and overlap a lot. This indicates that the chains are mixing well and not overly affected by their starting point.

8.4.3 Running chains in parallel

Although this model runs very quickly, others models may take considerably longer. We can use `jags.parallel()` in place of `jags()` to take advantage of multiple cores to run more than one chain at a time. `jags.seed` can be used to set the seed for the parallel random number generator used. (Note: `set.seed()` does not work when using `jags.parallel()` and `jags.seed` has no effect when using `jags()`.)

```
library(R2jags)
bern_jags5 <-
  jags.parallel(
    data = list(y = z15N50$y, N = nrow(z15N50)),
    model.file = bern_model,
    parameters.to.save = c("theta"),
    n.chains = 4, n.iter = 5000, n.burnin = 1000,
    jags.seed = 12345
  )
```

8.5 Example 2: comparing two proportions

We have seen this situation before when we compared two coins. This time we'll be a little more personal and compare two people. We will also work with a data set in a slightly different form. But the main point will be to see how we describe this familiar model to JAGS.

8.5.1 The data

Suppose we want to compare Reginald and Tony's abilities to hit a target (with a dart, perhaps). For each attempt, we record two pieces of information: the person making the attempt (the subject) and whether the attempt succeeded (0 or 1).

Kruschke's provides a data frame for this, but the names he uses are not good practice, so let's remanme them to be more like what you might see in a real data set.

```
library(mosaic)
head(z6N8z2N7)
```

y	s
1	Reginald
0	Reginald
1	Reginald
1	Reginald
1	Reginald
1	Reginald

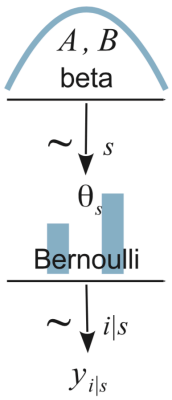
```
# Let's do some renaming
Target <- z6N8z2N7 %>%
  rename(hit = y, subject = s)
df_stats(hit ~ subject, data = Target, props, attempts = length)
```

subject	prop_0	prop_1	attempts
Reginald	0.2500	0.7500	8
Tony	0.7143	0.2857	7

Reginald was more successful than Tony, but neither had very many attempts.

8.5.2 The model

Now our model is that each person has his own success rate – we have **two θ's**, one for Reginald and one for Tony.



We express this as

$$Y_i|s \sim \text{Bern}(\theta_s)$$
$$\theta_s \sim \text{Beta}(a, b)$$

8.5.3 Describing the model to JAGS

```
bern2_model <- function() {
  for (i in 1:Nobs) {
    # each response is Bernoulli with the appropriate theta
    hit[i] ~ dbern(theta[subject[i]])
  }
  for (s in 1:Nsub) {
    theta[s] ~ dbeta(2, 2)    # prior for each theta
  }
}
```

JAGS will also need access to four pieces of information from our data set:

- a vector of `hit` values
- a vector of `subject` values – coded as integers 1 and 2 (so that `subject[i]` makes sense to JAGS. (In general, JAGS is much less fluid in handling data than R is, so we often need to do some manual data conversion for JAGS.)
- `Nobs` – the total number of observations
- `Nsub` – the number of subjects

We will prepare these as a list.

```
TargetList <-
  list(
    Nobs = nrow(Target),
    Nsub = 2,
    hit = Target$hit,
    subject = as.numeric(as.factor(Target$subject))
)
TargetList
```

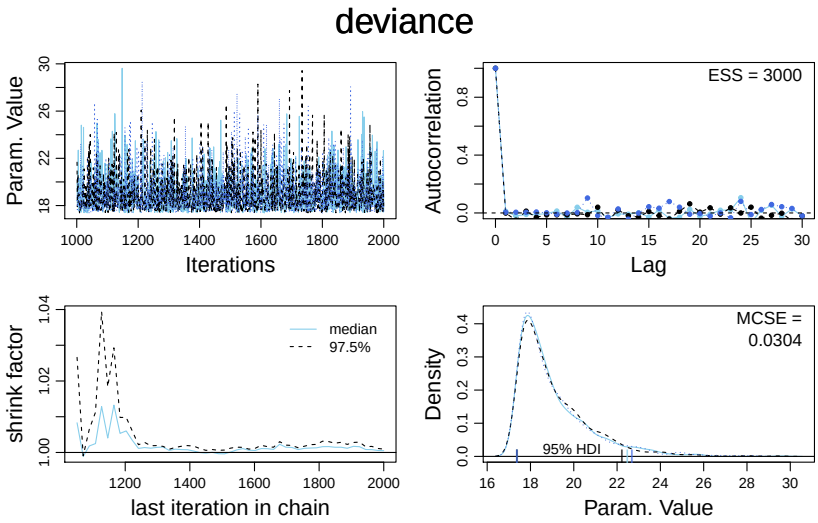
```
## $Nobs
## [1] 15
##
## $Nsub
## [1] 2
##
## $hit
##  [1] 1 0 1 1 1 1 1 0 0 0 1 0 0 1 0
##
## $subject
##  [1] 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2
```

8.5.4 Fitting the model

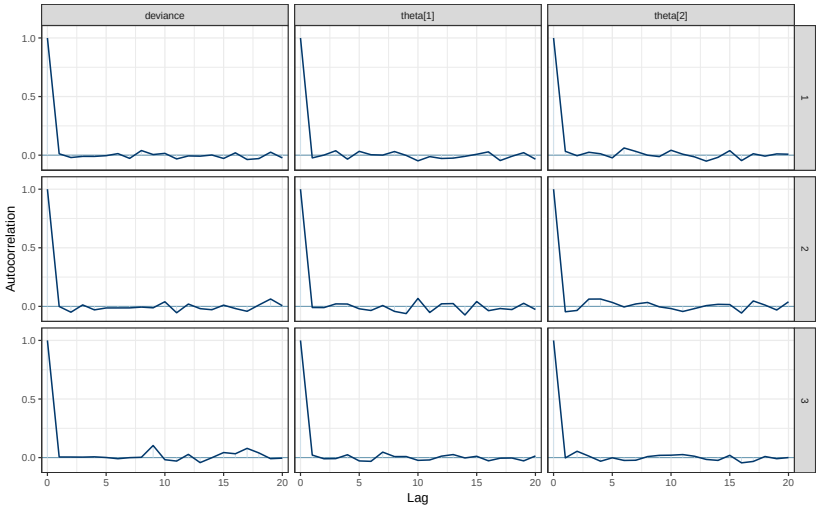
```
bern2_jags <-
  jags(
    data = TargetList,
    model = bern2_model,
    parameters.to.save = "theta")
```

8.5.5 Inspecting the results

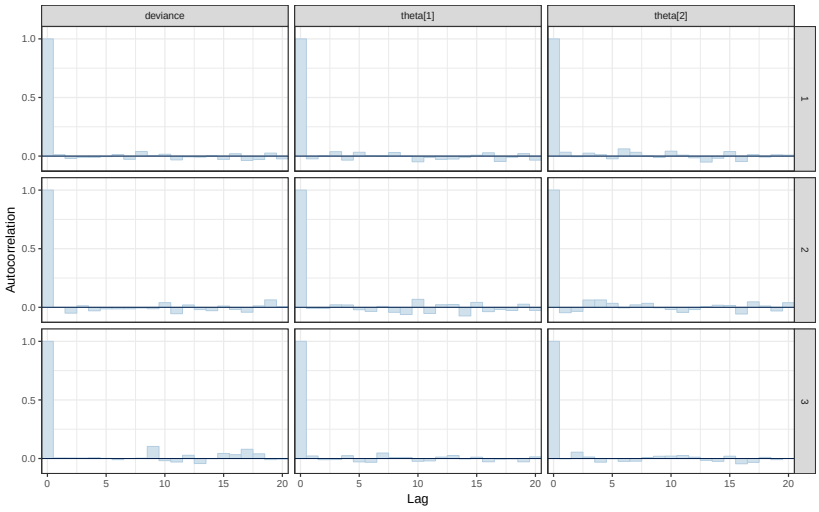
```
bern2_mcmc <- as.mcmc(bern2_jags)
# Kruschke diagnostic plots
diag_mcmc(bern2_mcmc)
```



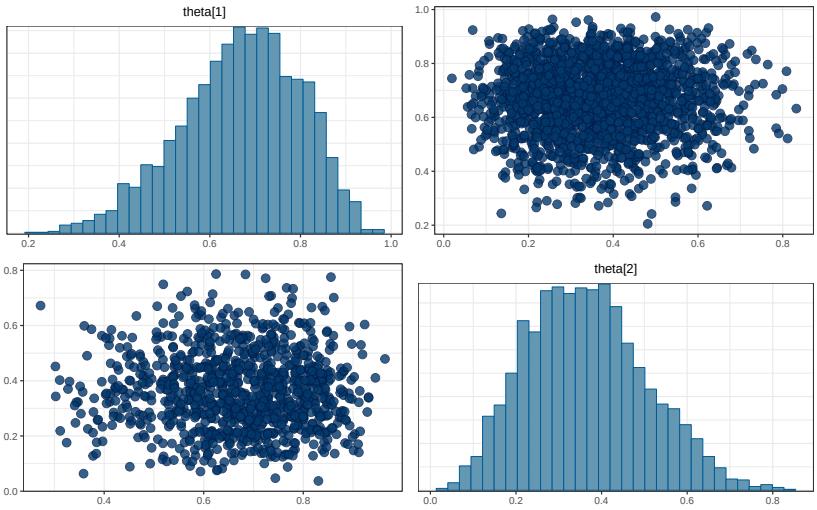

```
# bayesplot plots
mcmc_acf(bern2_mcmc)
```



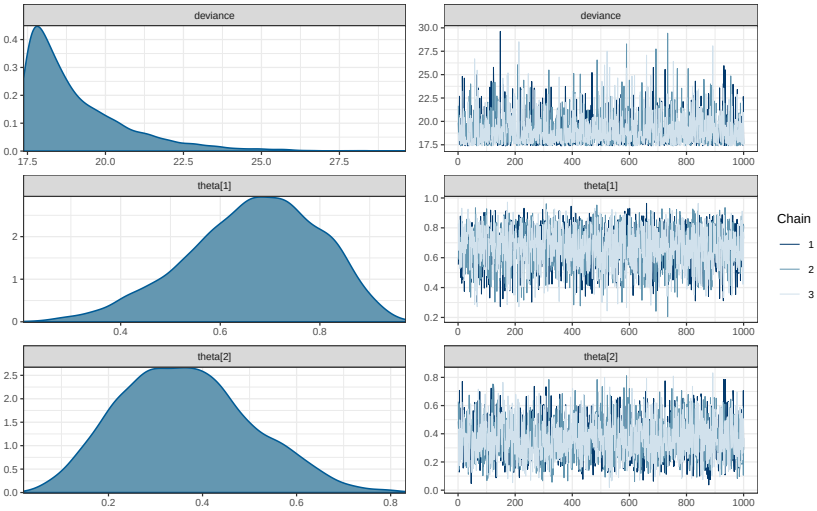

```
mcmc_acf_bar(bern2_mcmc)
```



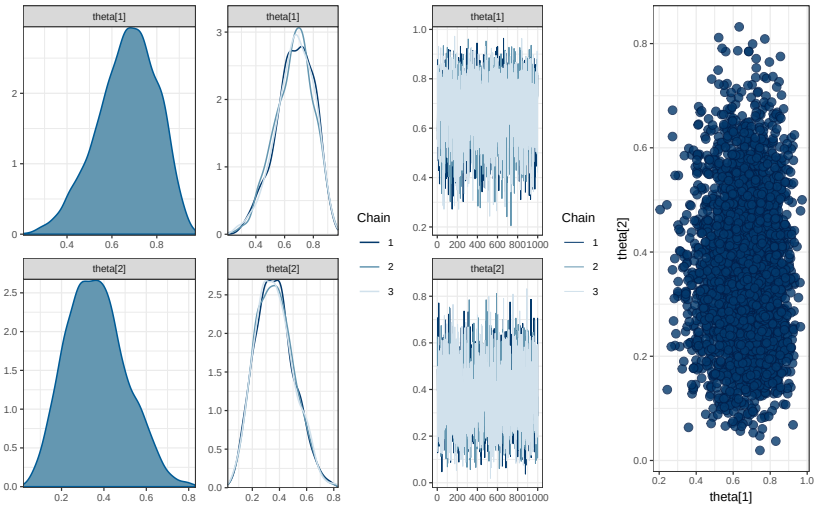
```
mcmc_pairs(bern2_mcmc, pars = c("theta[1]", "theta[2]"))
```




```
mcmc_combo(bern2_mcmc)
```




```
mcmc_combo(bern2_mcmc, combo = c("dens", "dens_overlay", "trace", "scatter"),
           pars = c("theta[1]", "theta[2]"))
```



Here is a list of `mcmc_` functions available:

```
apropos("^mcmc_")
```

```
##  [1] "mcmc_acf"                "mcmc_acf_bar"
##  [3] "mcmc_areas"              "mcmc_areas_data"
##  [5] "mcmc_areas_ridges"       "mcmc_areas_ridges_data"
##  [7] "mcmc_combo"              "mcmc_dens"
##  [9] "mcmc_dens_chains"        "mcmc_dens_chains_data"
## [11] "mcmc_dens_overlay"       "mcmc_hex"
## [13] "mcmc_hist"               "mcmc_hist_by_chain"
## [15] "mcmc_intervals"          "mcmc_intervals_data"
## [17] "mcmc_neff"               "mcmc_neff_data"
## [19] "mcmc_neff_hist"          "mcmc_nuts_acceptance"
## [21] "mcmc_nuts_divergence"    "mcmc_nuts_energy"
## [23] "mcmc_nuts_stepsize"      "mcmc_nuts_treedepth"
## [25] "mcmc_pairs"              "mcmc_parcoord"
## [27] "mcmc_parcoord_data"      "mcmc_recover_hist"
## [29] "mcmc_recover_intervals" "mcmc_recover_scatter"
## [31] "mcmc_rhat"               "mcmc_rhat_data"
## [33] "mcmc_rhat_hist"          "mcmc_scatter"
## [35] "mcmc_trace"              "mcmc_trace_highlight"
## [37] "mcmc_violin"
```

The functions ending in `_data()` return the data used to make the corresponding plot. This can be useful if you want to display that same information in a different way or if you just want to inspect the data to make sure you understand the plot.

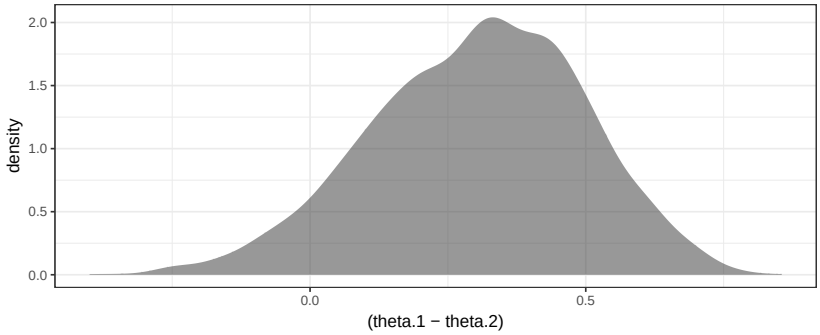
8.5.6 Difference in proportions

If we are primarily interested in the difference between Reginald and Tony, we can plot the difference in their theta values.

```
head(posterior(bern2_jags))
```

deviance	theta.1	theta.2	chain	iter
19.88	0.5540	0.4877	chain:1	1
18.18	0.6395	0.3935	chain:1	2
19.09	0.7375	0.5295	chain:1	3
19.55	0.5836	0.1285	chain:1	4
22.69	0.7354	0.7078	chain:1	5
17.53	0.7893	0.2377	chain:1	6

```
gf_density( ~(theta.1 - theta.2), data = posterior(bern2_jags))
```



8.5.7 Sampling from the prior

To sample from the prior, we must do the following:

- remove the response variable from our data list
- change `Nobs` to 0
- set `DIC = FALSE` in the call to `jags()`.

This will run the model without any data, which means the posterior will be the same as the prior.

```
# make a copy of our data list
TargetList0 <- list(
  Nobs = 0,
  Nsub = 2,
  subject = as.numeric(as.factor(Target$subject))
)

bern2_jags0 <-
  jags(
    data = TargetList0,
    model.file = bern2_model,
    parameters.to.save = c("theta"),
    n.chains = 2, n.iter = 5000, n.burnin = 1000,
    DIC = FALSE)
```

```
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 0
##    Unobserved stochastic nodes: 2
##    Total graph size: 20
##
## Initializing model
```

8.5.7.1 Note about : in JAGS and in R

From the JAGS documentation:

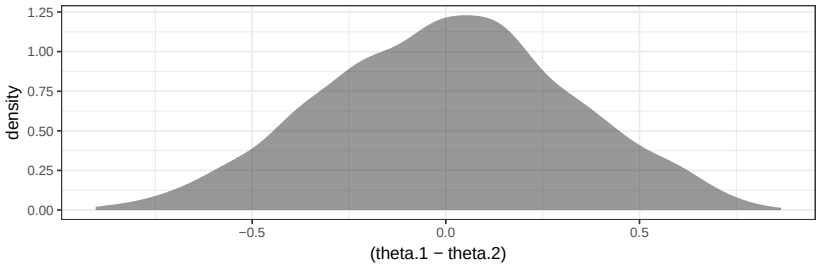
> The sequence operator `:` can only produce increasing sequences. If n < m then `m:n` produces a vector of length zero and when this is used in a for loop index expression the contents of loop inside the curly brackets are skipped. Note that this behavior is different from the sequence operator in R, where `m:n` will produce a decreasing sequence if `n < m`.

So in our JAGS model, `1:0` correctly represents no data (and no trips through the for loop).

8.5.7.2 What good is it to generate samples from the prior?

Our model set priors for θ_1 and θ_2, but this implies a distribution for $\theta_1 - \theta_2$, and we might like to see what that distribution looks like.

```
gf_density( ~(theta.1 - theta.2), data = posterior(bern2_jags0))
```



8.6 Exercises

1. **Sampling from priors.** You want to know who is the better free throw shooter, Alice or Bob. You decide to have each shoot a number of shots and record their makes and misses. You are primarily interested in the difference between their free throw shooting proportions ($\theta_2 - \theta_1$), and you are curious to know how your choice of priors for θ_1 and θ_2 affects the prior for $\theta_2 - \theta_1$. For each situation below, use JAGS to sample from the prior distribution for $\theta_2 - \theta_1$ and create a density plot. (In each case, assume the priors for θ_1 and θ_2 are independent.)

   a. Both priors are uniform. What distribution do you think the prior for $\theta_2 - \theta_1$ is?

   b. Both priors are $\mathsf{Beta}(0.2, 0.2)$. Explain why the prior for $\theta_2 - \theta_1$ looks the way it does.

   Hint: Don't forget to set `DIC = FALSE` when sampling from the prior.

2. Now suppose that Alice makes 25 out of 30 shots and Bob makes 18 out of 32. Is this enough evidence to conclude that Alice is the better shooter? Do this two ways. In each case, use a $\mathsf{Beta}(4, 2)$ prior for both θ_1 and θ_2.

   a. Create data and use a model like the one used elsewhere in this chapter. [Hint: `rep()` is handy for creating a bunch of values that are the same.]

   b. Instead of using `dbern()`, use `dbin()` (JAGS version of the binomial distribution). This should allow you to get by with simpler data that consists only of the numbers 25, 30, 18, and 32. Note: the order of arguments for `dbin()` is probability first, then number of trials. This is reversed from the order in R.

3. In the previous problem, what does this prior say about the beliefs about Alice and Bob's shooting before gathering data?

4. Let's think about Alice and Bob some more. We don't know how to do this yet, but explain why if you knew very little about basketball, you might like to have a prior for θ_1 and θ_2 that was not indpendent. How might the priors for θ_1 and θ_2 be related?

5. Consider the model below.

   a. Create a plot of the **prior** distribution of `theat1` and `theta2`. A scatter plot with overlayed density plot works well. How does this prior compare to the priors in problem 1. (Hint: Don't forget to use `DIC = FALSE` when sampling from the prior.)

   b. Fit the model to the Alice and Bob data. How does this choice of prior change things?

```
diff_model <- function() {
  z1 ~ dbin(theta1, n1)
  z2 ~ dbin(theta2, n2)
  theta2 <- theta1 + delta
  theta1 ~ dbeta(4, 2)
  delta ~ dnorm(0, 400)   # Normal with mean 0 and sd = 0.05
}

diff_jags <-
  jags.parallel(
    model = diff_model,
    data  = list(n1 = 30, z1 = 25, n2 = 32, z2 = 18),
    parameters.to.save = c("theta1", "theta2", "delta"),
    n.iter = 5000,
    n.chains = 4
  )
```

6. Redo problem 5 using the grid method.

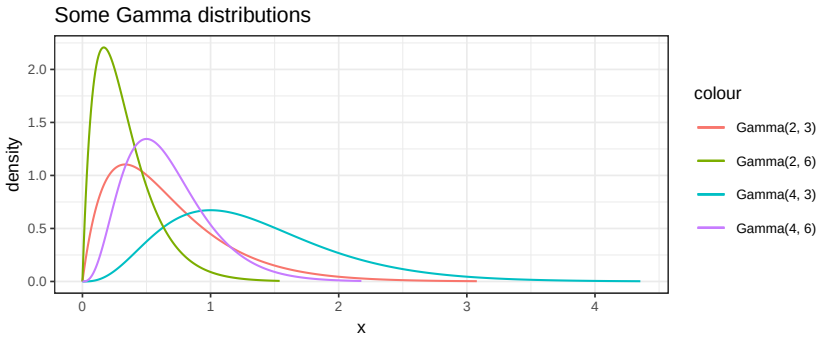
Chapter 9

Heierarchical Models

9.1 Gamma Distributions

We will use gamma distributions for some of our priors in this chapter. Gamma distributions have support $(0, \infty)$ and are skewed to the right. Both R and JAGS paramterize the gamma distributions with two parameters called shape and rate.

```
gf_dist("gamma", shape = 2, rate = 3, color = ~"Gamma(2, 3)") %>%
gf_dist("gamma", shape = 4, rate = 3, color = ~"Gamma(4, 3)") %>%
gf_dist("gamma", shape = 2, rate = 6, color = ~"Gamma(2, 6)") %>%
gf_dist("gamma", shape = 4, rate = 6, color = ~"Gamma(4, 6)") %>%
  gf_labs(title = "Some Gamma distributions")
```

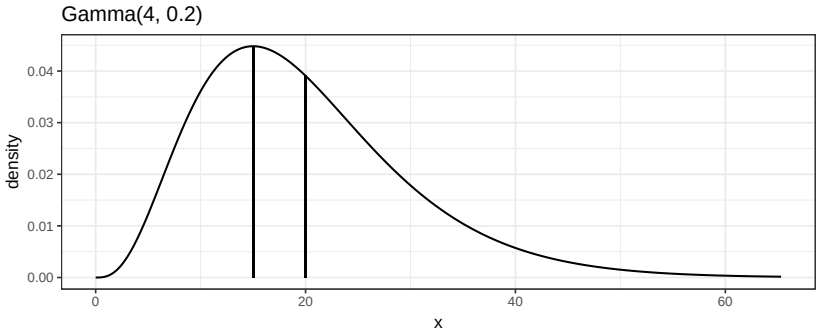


The mean, mode, standard deviation can be calcuated from the shape s and rate r as follows:

$$\mu = \frac{s}{r}$$
$$\omega = \frac{s1}{r} \qquad (s > 1)$$
$$\sigma = \frac{\sqrt{s}}{r}$$

In addition, the scale parameter (1/rate) is sometimes used in place of the rate parameter. The `gamma_params()` function will automate conversion between various parameterizations. It works just like `beta_params()` that we have seen before.

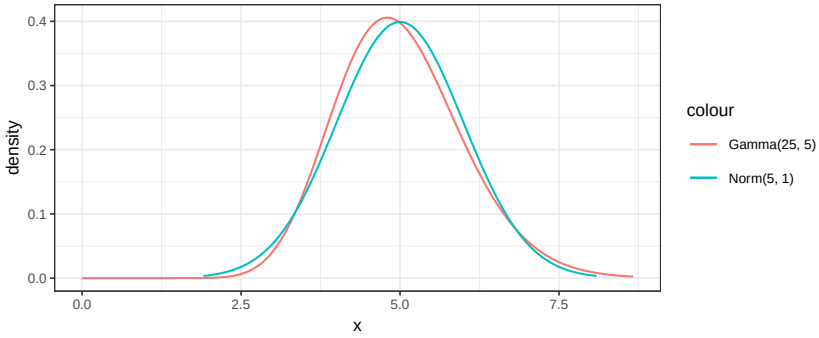
```
gamma_params(mode = 15, sd = 10, plot = TRUE)
```



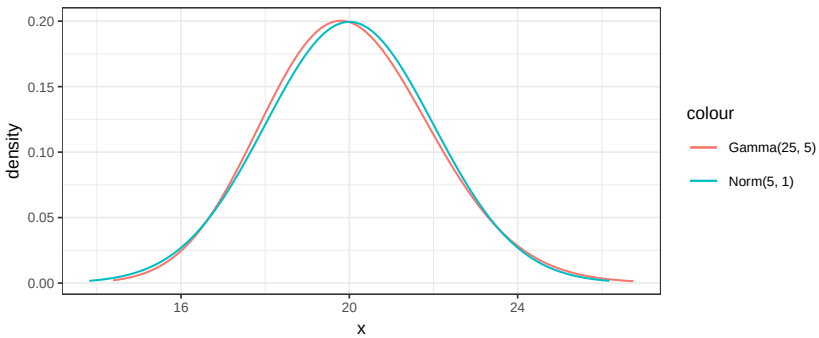
shape	rate	scale	mode	mean	sd
4	0.2	5	15	20	10

As the shape parameter gets larger and larger, the gamma distribution becomes less and less skewed (and more and more like a normal distribution):

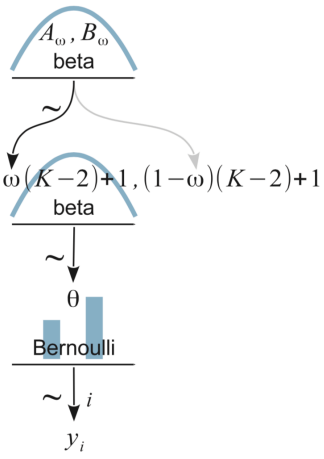
```
gf_dist("gamma", shape = 25, rate = 5, color = ~"Gamma(25, 5)") %>%
  gf_dist("norm", mean = 5, sd = 1, color = ~"Norm(5, 1)")
```



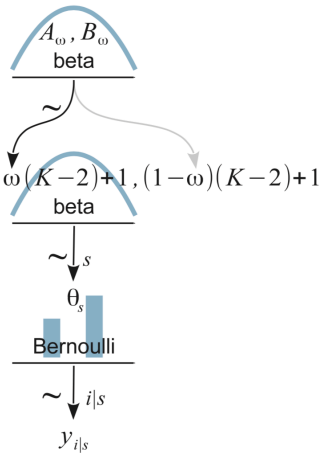

```
gf_dist("gamma", shape = 100, rate = 5, color = ~"Gamma(25, 5)") %>%
  gf_dist("norm", mean = 20, sd = 2, color = ~"Norm(5, 1)")
```



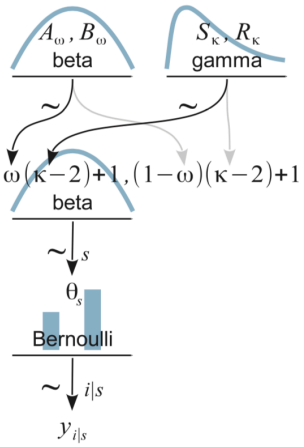
9.2 One coin from one mint



9.3 Multiple coins from one mint



9.4 Multiple coins from multiple mints



9.5 Therapeutic Touch

The study is described in the text. The article reporting on the study can be found at https://jamanetwork.com/journals/jama/fullarticle/187390. Here's the abstract:

9.5.1 Abstract

Context.— Therapeutic Touch (TT) is a widely used nursing practice rooted in mysticism but alleged to have a scientific basis. Practitioners of TT claim to treat many medical conditions by using their hands to manipulate a "human energy field" perceptible above the patient's skin.

Objective.— To investigate whether TT practitioners can actually perceive a "human energy field."

Design.— Twenty-one practitioners with TT experience for from 1 to 27 years were tested under blinded conditions to determine whether they could correctly identify which of their hands was closest to the investigator's hand. Placement of the investigator's hand was determined by flipping a coin. Fourteen practitioners were tested 10 times each, and 7 practitioners were tested 20 times each.

Main Outcome Measure.— Practitioners of TT were asked to state whether the investigator's unseen hand hovered above their right hand or their left hand. To show the validity of TT theory, the practitioners should have been able to locate the investigator's hand 100% of the time. A score of 50% would be expected through chance alone.

Results.— Practitioners of TT identified the correct hand in only 123 (44%) of 280 trials, which is close to what would be expected for random chance. There was no significant correlation between the practitioner's score and length of experience (r=0.23). The statistical power of this experiment was sufficient to conclude that if TT practitioners could reliably detect a human energy field, the study would have demonstrated this.

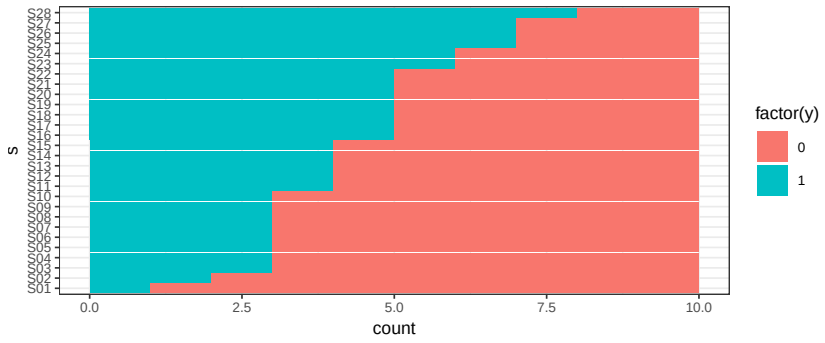
Conclusions.— Twenty-one experienced TT practitioners were unable to detect the investigator's "energy field." Their failure to substantiate TT's most fundamental claim is unrefuted evidence that the claims of TT are groundless and that further professional use is unjustified.

9.5.2 Data

```
library(mosaic)
head(TherapeuticTouch, 3)
```

y	s
1	S01
0	S01
0	S01

```
gf_barh(s ~ ., data = TherapeuticTouch, fill = ~ factor(y))
```



9.5.3 A heierarchical model

Big ideas:

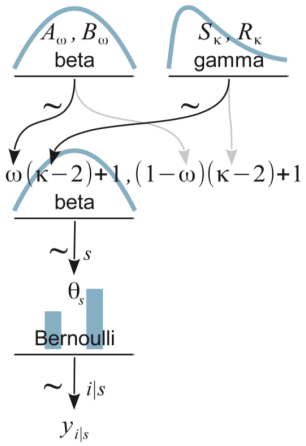
- The ten trials for each subject are a sample from the many trials that could have been done.
  - distribution of results: $\mathsf{Bern}(\theta_s)$ – each subject has a potentially different θ_s.
- The subjects themselves are just a sample from all of the TT practitioners that could have been in the study.
  - So the θ_s values are a sample from a distribution of θ values for all TT practititioners and tell us something about that distribution.
  - We will assume a beta distribution for this, where the parameters are unknown and estimated from the data.
- Use the data to estimate both the individual level θ_s values and the group level parameters of the beta distribution.
- Parameterization of the beta distribution for θ_s.
  - We are primarily interested in the mean or mode of this distribution (typical value of θ for TT practitioner).
  - Many combinations of shape parameters give the same mean (or mode), and they are highly correlated. For example, $\mathsf{Beta}(2, 4)$, $\mathsf{Beta}(20, 40)$, and $\mathsf{Beta}(200, 400)$ all have a mean of $1/3$.
  - We will parameterize this Beta distribution with **mode** (ω), and **concentration** (κ)
  - We will need to convert mode and concentration into the two shape parameters, since JAGS and R use the two shape parameters.

$$\alpha = \omega(\kappa - 2) + 1 \tag{9.1}$$
$$\beta = (1 - \omega)(\kappa - 2) + 1) \tag{9.2}$$

- ω and κ will need priors

- ω: Beta
- $\kappa - 2$: Gamma (because $\kappa > 2$)

Putting this altogether we have the following picture:



Now we code it up for JAGS.

```
gamma_params(mean = 1, sd = 10)
```

shape	rate	scale	mode	mean	sd
0.01	0.01	100	NA	1	10

```
touch_model <- function() {
   for (i in 1:Ntotal) {
     y[i] ~ dbern(theta[s[i]])
   }
  for (s in 1:Nsubj) {
    theta[s] ~ dbeta(omega * (kappa - 2) + 1, (1 - omega) * (kappa - 2) + 1)
  }
  omega ~ dbeta(1, 1)
  kappa <- kappaMinusTwo + 2
  kappaMinusTwo ~ dgamma(0.01, 0.01)      # mean = 1, sd = 10
  }
```

```
set.seed(1234)
TouchData <- list(
  Ntotal = nrow(TherapeuticTouch),
  Nsubj = length(unique(TherapeuticTouch$s)),
  y = TherapeuticTouch$y,
  # must convert subjects to sequence 1:Nsubj
  s = as.numeric(factor(TherapeuticTouch$s))
)
touch_jags <-
  jags(
  data = TouchData,
  model = touch_model,
  parameters.to.save = c("theta", "kappa", "omega"),
)
```

```
touch_jags
```

```
## Inference for Bugs model at "/var/folders/py/txwd26jx5rq83f4nn0f5fmmm0000gn/T//RtmpkAYhBH/model1121d663
```

```
##  3 chains, each with 2000 iterations (first 1000 discarded)
##  n.sims = 3000 iterations saved
##           mu.vect sd.vect    2.5%     25%     50%     75%   97.5%  Rhat
## kappa      52.286  54.028   8.932  20.025  34.399  65.489 210.966 1.056
## omega       0.439   0.035   0.367   0.416   0.439   0.462   0.508 1.004
## theta[1]    0.363   0.088   0.175   0.305   0.369   0.426   0.517 1.016
## theta[2]    0.384   0.084   0.208   0.331   0.389   0.444   0.536 1.015
## theta[3]    0.409   0.081   0.241   0.358   0.414   0.463   0.564 1.005
## theta[4]    0.411   0.083   0.236   0.357   0.415   0.467   0.564 1.020
## theta[5]    0.409   0.080   0.244   0.358   0.411   0.461   0.563 1.010
## theta[6]    0.410   0.082   0.235   0.359   0.414   0.464   0.562 1.006
## theta[7]    0.409   0.080   0.244   0.359   0.412   0.461   0.560 1.019
## theta[8]    0.410   0.080   0.243   0.360   0.413   0.462   0.565 1.012
## theta[9]    0.407   0.080   0.247   0.356   0.410   0.459   0.561 1.008
## theta[10]   0.409   0.081   0.238   0.358   0.413   0.463   0.559 1.008
## theta[11]   0.433   0.077   0.276   0.384   0.433   0.483   0.588 1.004
## theta[12]   0.433   0.081   0.278   0.383   0.434   0.485   0.593 1.005
## theta[13]   0.434   0.078   0.280   0.385   0.434   0.484   0.587 1.002
## theta[14]   0.432   0.080   0.272   0.380   0.432   0.482   0.593 1.001
## theta[15]   0.433   0.080   0.275   0.381   0.434   0.484   0.590 1.004
## theta[16]   0.457   0.080   0.300   0.408   0.454   0.504   0.624 1.002
## theta[17]   0.456   0.079   0.295   0.406   0.453   0.506   0.621 1.002
## theta[18]   0.454   0.081   0.295   0.401   0.453   0.504   0.626 1.003
## theta[19]   0.457   0.078   0.313   0.407   0.454   0.505   0.618 1.001
## theta[20]   0.455   0.080   0.298   0.403   0.452   0.508   0.612 1.003
## theta[21]   0.457   0.082   0.296   0.404   0.455   0.510   0.621 1.005
## theta[22]   0.457   0.079   0.311   0.405   0.454   0.507   0.625 1.003
## theta[23]   0.480   0.083   0.328   0.425   0.474   0.529   0.658 1.008
## theta[24]   0.482   0.083   0.327   0.427   0.476   0.531   0.660 1.003
## theta[25]   0.506   0.088   0.351   0.446   0.497   0.562   0.691 1.008
## theta[26]   0.507   0.088   0.354   0.446   0.498   0.560   0.700 1.003
## theta[27]   0.505   0.087   0.355   0.445   0.495   0.559   0.695 1.008
## theta[28]   0.527   0.093   0.374   0.460   0.517   0.581   0.735 1.008
## deviance  378.963   5.156 368.591 375.437 379.284 382.411 388.755 1.010
##           n.eff
## kappa        44
## omega      1100
## theta[1]    210
## theta[2]    190
## theta[3]    970
## theta[4]    270
## theta[5]    500
## theta[6]   1100
## theta[7]    180
## theta[8]    510
## theta[9]    650
## theta[10]   710
## theta[11]  1700
## theta[12]   710
## theta[13]  3000
## theta[14]  3000
## theta[15]  3000
## theta[16]  3000
## theta[17]  1600
```

```
## theta[18]  3000
## theta[19]  3000
## theta[20]  3000
## theta[21]  1000
## theta[22]  1800
## theta[23]   510
## theta[24]  1300
## theta[25]   260
## theta[26]   680
## theta[27]   250
## theta[28]   300
## deviance    220
##
## For each parameter, n.eff is a crude measure of effective sample size,
## and Rhat is the potential scale reduction factor (at convergence, Rhat=1).
##
## DIC info (using the rule, pD = var(deviance)/2)
## pD = 13.2 and DIC = 392.1
## DIC is an estimate of expected predictive error (lower deviance is better).
```

What do we learn from a quick look at this output?

- The Rhat values look good
- The autocorrelation varies from parameter to parameter. For some parameters, it looks like it will take a much longer run to get a large effective sample size.

So let's do a larger run.

```
touch_jags <-
  jags.parallel(
  data = TouchData,
  model = touch_model,
  parameters.to.save = c("theta", "kappa", "omega"),
  n.burnin = 1000,
  n.iter = 41000,
  n.chains = 5,
  n.thin = 10,
  jags.seed = 54321
)
```

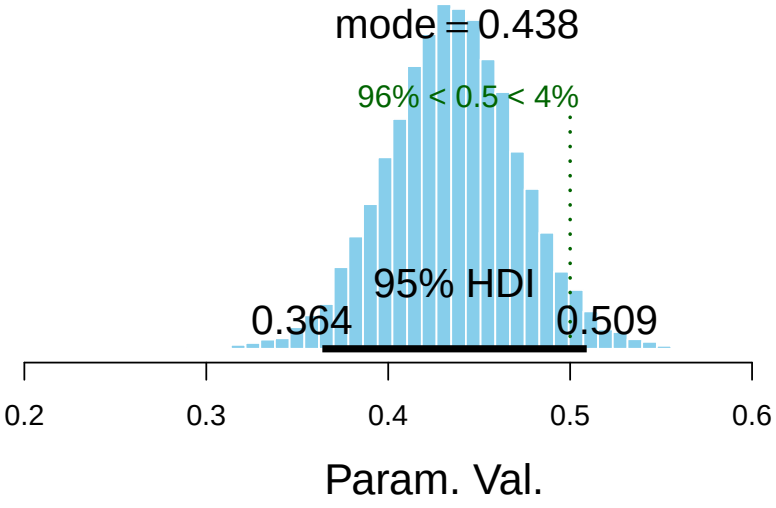
```
touch_jags
```

```
## Inference for Bugs model at "touch_model", fit using jags,
##  5 chains, each with 41000 iterations (first 1000 discarded), n.thin = 10
##  n.sims = 20000 iterations saved
##           mu.vect sd.vect    2.5%     25%     50%     75%   97.5%  Rhat
## kappa      55.710  55.745   8.632  21.295  37.091  68.780 210.357 1.001
## omega       0.435   0.037   0.362   0.412   0.436   0.460   0.508 1.001
## theta[1]    0.360   0.087   0.169   0.305   0.367   0.422   0.513 1.001
## theta[2]    0.384   0.084   0.206   0.332   0.389   0.441   0.536 1.001
## theta[3]    0.407   0.080   0.240   0.357   0.410   0.460   0.560 1.001
## theta[4]    0.407   0.080   0.239   0.357   0.410   0.461   0.560 1.001
## theta[5]    0.408   0.080   0.240   0.359   0.411   0.460   0.561 1.001
## theta[6]    0.407   0.080   0.240   0.357   0.410   0.459   0.561 1.001
## theta[7]    0.407   0.080   0.241   0.358   0.410   0.459   0.559 1.001
## theta[8]    0.407   0.080   0.238   0.357   0.411   0.459   0.560 1.001
## theta[9]    0.408   0.080   0.241   0.358   0.411   0.460   0.563 1.001
```

```
## theta[10]   0.408   0.080   0.242   0.358   0.411   0.460   0.563 1.001
## theta[11]   0.431   0.079   0.274   0.382   0.431   0.481   0.591 1.001
## theta[12]   0.430   0.079   0.271   0.380   0.430   0.480   0.588 1.001
## theta[13]   0.431   0.079   0.272   0.381   0.430   0.481   0.588 1.001
## theta[14]   0.430   0.078   0.276   0.380   0.430   0.479   0.590 1.001
## theta[15]   0.431   0.078   0.274   0.382   0.432   0.481   0.590 1.001
## theta[16]   0.454   0.080   0.300   0.402   0.451   0.503   0.622 1.001
## theta[17]   0.454   0.080   0.303   0.402   0.451   0.503   0.624 1.001
## theta[18]   0.455   0.080   0.301   0.403   0.451   0.505   0.624 1.001
## theta[19]   0.454   0.080   0.301   0.402   0.452   0.502   0.626 1.001
## theta[20]   0.454   0.079   0.300   0.403   0.452   0.502   0.620 1.001
## theta[21]   0.455   0.080   0.303   0.403   0.452   0.504   0.621 1.001
## theta[22]   0.455   0.080   0.302   0.402   0.451   0.504   0.622 1.001
## theta[23]   0.477   0.083   0.327   0.422   0.471   0.527   0.659 1.001
## theta[24]   0.476   0.082   0.328   0.422   0.470   0.526   0.657 1.001
## theta[25]   0.501   0.087   0.349   0.441   0.492   0.553   0.691 1.001
## theta[26]   0.500   0.086   0.351   0.441   0.493   0.551   0.691 1.001
## theta[27]   0.500   0.086   0.350   0.441   0.493   0.553   0.691 1.001
## theta[28]   0.525   0.093   0.370   0.460   0.514   0.581   0.733 1.001
## deviance  379.089   5.187 368.486 375.671 379.322 382.640 388.787 1.001
##           n.eff
## kappa      8100
## omega      8900
## theta[1]  19000
## theta[2]  20000
## theta[3]  20000
## theta[4]  19000
## theta[5]   8400
## theta[6]  20000
## theta[7]   8100
## theta[8]  12000
## theta[9]   6900
## theta[10] 13000
## theta[11] 20000
## theta[12] 20000
## theta[13] 10000
## theta[14] 20000
## theta[15] 20000
## theta[16] 20000
## theta[17] 20000
## theta[18] 20000
## theta[19] 20000
## theta[20] 20000
## theta[21] 20000
## theta[22] 18000
## theta[23] 12000
## theta[24] 15000
## theta[25]  9700
## theta[26] 20000
## theta[27]  9900
## theta[28] 20000
## deviance  20000
##
## For each parameter, n.eff is a crude measure of effective sample size,
```

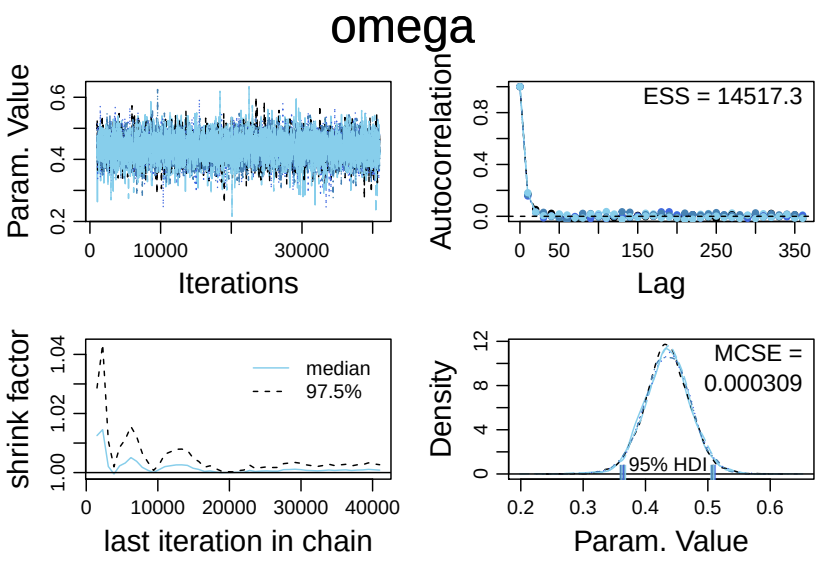
```
## and Rhat is the potential scale reduction factor (at convergence, Rhat=1).
##
## DIC info (using the rule, pD = var(deviance)/2)
## pD = 13.5 and DIC = 392.5
## DIC is an estimate of expected predictive error (lower deviance is better).
```

```
touch_mcmc <- as.mcmc(touch_jags)
plot_post(touch_mcmc[, "omega"], comparison_value = 0.5)
```

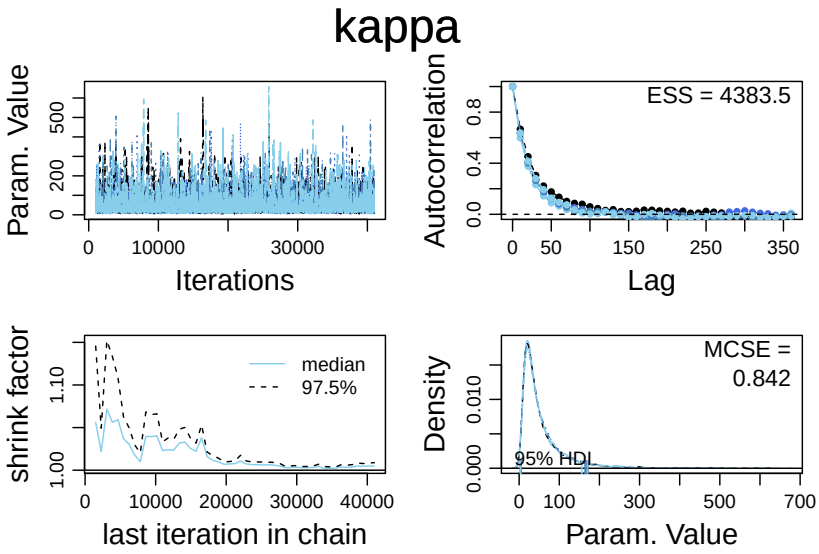



```
## $posterior
##        ESS   mean median   mode
## var1 14724 0.4354 0.4357 0.4379
##
## $hdi
##   prob     lo     hi
## 1 0.95 0.3638 0.5091
##
## $comparison
##   value P(< comp. val.) P(> comp. val.)
## 1   0.5          0.9597         0.04025
```

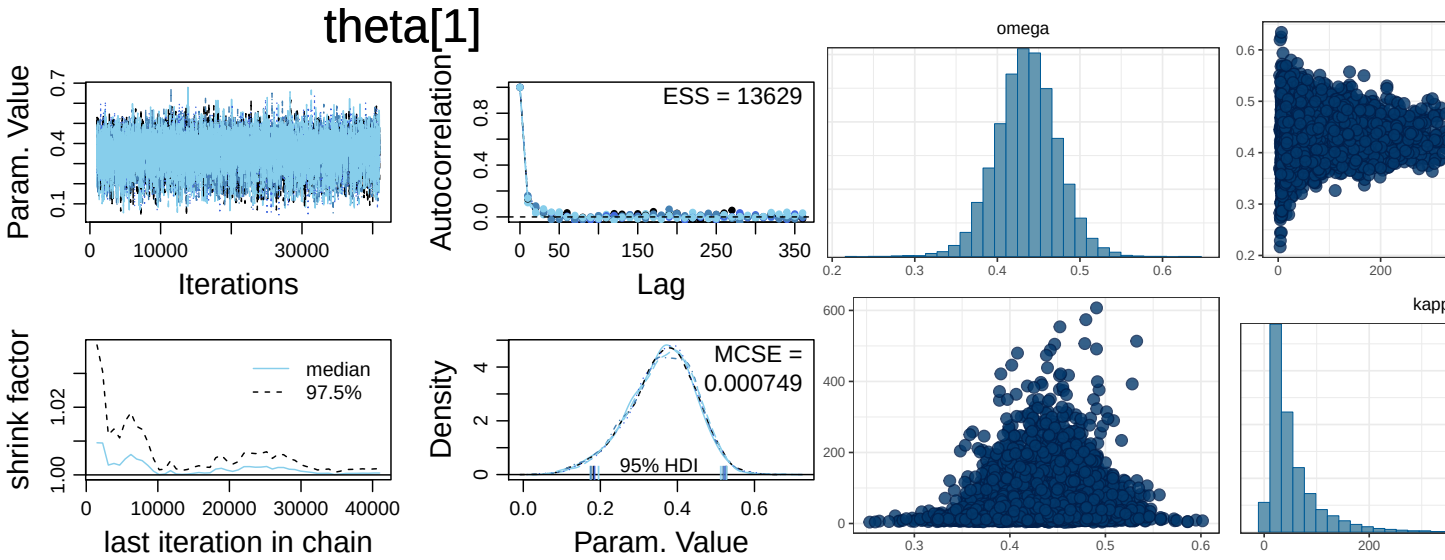
```
diag_mcmc(touch_mcmc, par = "omega")
```



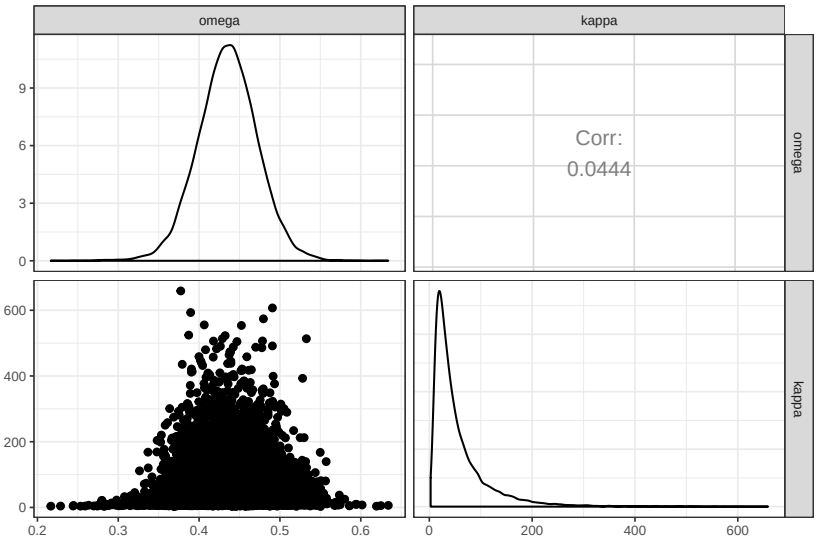
```
diag_mcmc(touch_mcmc, par = "kappa")
```



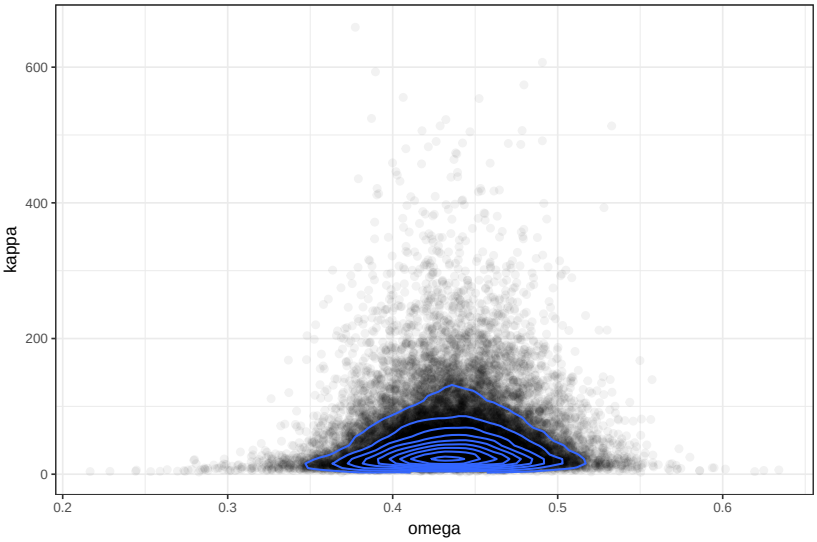

```
diag_mcmc(touch_mcmc, par = "theta[1]")
mcmc_pairs(touch_mcmc, pars = c("omega", "kappa"))
```




```
GGally::ggpairs(posterior(touch_jags) %>% select(omega, kappa))
```




```
gf_point(kappa ~ omega, data = posterior(touch_jags), alpha = 0.05) %>%
  gf_density2d(kappa ~ omega, data = posterior(touch_jags))
```



9.6 Other parameterizations we might have tried

9.6.1 Shape parameters for Beta

Suppose we decided to parameterize the beta distribution with shape parameters like this?

```
touch_model2 <- function() {
   for (i in 1:Ntotal) {
     y[i] ~ dbern(theta[s[i]])
   }
  for (s in 1:Nsubj) {
    theta[s] ~ dbeta(alpha, beta)
  }
  kappa <- alpha + beta
  mu <- alpha / (alpha + beta)
```

```
  alpha <- alphaMinusOne + 1
  beta  <- betaMinusOne + 1
  alphaMinusOne ~ dgamma(0.01, 0.01)
  betaMinusOne ~ dgamma(0.01, 0.01)
  }
```

We'll run it with the same options we used above to faciliate easy comparisons.

```
touch_jags2 <-
  jags.parallel(
  data = TouchData,
  model = touch_model2,
  parameters.to.save = c("theta", "alpha", "beta", "mu", "omega", "kappa"),
  n.burnin = 1000,
  n.iter = 41000,
  n.chains = 5,
  n.thin = 10,
  jags.seed = 54321
)
```

The resuls are disasterous: Rhat values well above 1 and effective sample sizes that are *much* smaller than before.

```
touch_jags2
```

```
## Inference for Bugs model at "touch_model2", fit using jags,
##  5 chains, each with 41000 iterations (first 1000 discarded), n.thin = 10
##  n.sims = 20000 iterations saved
##           mu.vect sd.vect    2.5%     25%     50%     75%   97.5%  Rhat
## alpha      19.586  23.261   1.000   4.833  12.099  24.854  89.225 2.300
## beta       24.922  29.869   1.000   6.200  15.476  31.857 110.226 2.411
## kappa      44.508  52.940   2.000  11.184  27.621  56.852 197.948 2.369
## mu          0.448   0.040   0.369   0.420   0.447   0.482   0.506 1.157
## theta[1]    0.322   0.120   0.056   0.247   0.344   0.409   0.509 1.708
## theta[2]    0.355   0.107   0.112   0.292   0.370   0.431   0.532 1.435
## theta[3]    0.391   0.098   0.172   0.333   0.399   0.455   0.567 1.210
## theta[4]    0.391   0.097   0.175   0.335   0.400   0.455   0.570 1.205
## theta[5]    0.392   0.098   0.176   0.335   0.401   0.457   0.571 1.196
## theta[6]    0.391   0.098   0.170   0.334   0.400   0.456   0.570 1.213
## theta[7]    0.392   0.097   0.176   0.335   0.399   0.455   0.571 1.188
## theta[8]    0.391   0.097   0.174   0.334   0.399   0.456   0.567 1.204
## theta[9]    0.391   0.098   0.171   0.333   0.400   0.455   0.568 1.212
## theta[10]   0.391   0.098   0.175   0.333   0.399   0.455   0.568 1.205
## theta[11]   0.427   0.094   0.233   0.370   0.428   0.484   0.621 1.066
## theta[12]   0.427   0.094   0.233   0.371   0.429   0.485   0.619 1.064
## theta[13]   0.427   0.094   0.231   0.370   0.428   0.484   0.619 1.073
## theta[14]   0.428   0.094   0.229   0.371   0.429   0.485   0.619 1.071
## theta[15]   0.427   0.094   0.230   0.370   0.429   0.483   0.616 1.070
## theta[16]   0.463   0.096   0.286   0.402   0.456   0.517   0.679 1.046
## theta[17]   0.463   0.096   0.285   0.403   0.456   0.517   0.676 1.044
## theta[18]   0.464   0.096   0.286   0.403   0.457   0.518   0.682 1.048
## theta[19]   0.462   0.096   0.282   0.401   0.456   0.517   0.678 1.049
## theta[20]   0.463   0.095   0.288   0.402   0.457   0.517   0.677 1.047
## theta[21]   0.462   0.097   0.283   0.401   0.456   0.516   0.682 1.045
## theta[22]   0.463   0.096   0.287   0.401   0.456   0.518   0.680 1.048
## theta[23]   0.499   0.104   0.325   0.429   0.485   0.554   0.747 1.134
```

```
## theta[24]   0.500   0.104   0.325   0.430   0.486   0.555   0.747 1.135
## theta[25]   0.533   0.117   0.353   0.452   0.512   0.597   0.814 1.280
## theta[26]   0.533   0.117   0.354   0.450   0.513   0.597   0.812 1.282
## theta[27]   0.533   0.118   0.353   0.451   0.512   0.597   0.819 1.289
## theta[28]   0.569   0.133   0.373   0.471   0.540   0.644   0.880 1.459
## deviance  378.572   5.644 367.282 374.715 378.780 382.457 389.202 1.034
##           n.eff
## alpha         7
## beta          7
## kappa         7
## mu           27
## theta[1]     10
## theta[2]     14
## theta[3]     30
## theta[4]     30
## theta[5]     31
## theta[6]     30
## theta[7]     33
## theta[8]     30
## theta[9]     29
## theta[10]    31
## theta[11]   250
## theta[12]   260
## theta[13]   180
## theta[14]   220
## theta[15]   190
## theta[16]   260
## theta[17]   330
## theta[18]   270
## theta[19]   280
## theta[20]   260
## theta[21]   290
## theta[22]   280
## theta[23]    35
## theta[24]    35
## theta[25]    17
## theta[26]    17
## theta[27]    16
## theta[28]    12
## deviance    120
##
## For each parameter, n.eff is a crude measure of effective sample size,
## and Rhat is the potential scale reduction factor (at convergence, Rhat=1).
##
## DIC info (using the rule, pD = var(deviance)/2)
## pD = 15.4 and DIC = 394.0
## DIC is an estimate of expected predictive error (lower deviance is better).
```

9.6.2 Mean instead of mode

This change seems less dramatic. Let's see how using mean and concentraion compares to using mode and concentration.

```
touch_model3 <- function() {
   for (i in 1:Ntotal) {
     y[i] ~ dbern(theta[s[i]])
   }
  for (s in 1:Nsubj) {
    theta[s] ~ dbeta(mu * kappa, (1 - mu) * kappa)
  }
  mu ~ dbeta(2, 2)
  kappa <- kappaMinusTwo + 2
  kappaMinusTwo ~ dgamma(0.01, 0.01)
  }
```

```
touch_jags3 <-
  jags.parallel(
  data = TouchData,
  model = touch_model3,
  parameters.to.save = c("theta", "mu", "kappa"),
  n.burnin = 1000,
  n.iter = 41000,
  n.chains = 5,
  n.thin = 10,
  jags.seed = 54321
)
```

This model seems to perform reasonably well.

```
touch_jags3
```

```
## Inference for Bugs model at "touch_model3", fit using jags,
##  5 chains, each with 41000 iterations (first 1000 discarded), n.thin = 10
##  n.sims = 20000 iterations saved
##           mu.vect sd.vect    2.5%     25%     50%     75%   97.5%  Rhat
## kappa      58.519  58.419   9.258  22.258  39.046  72.852 222.144 1.001
## mu          0.441   0.033   0.377   0.419   0.441   0.463   0.507 1.001
## theta[1]    0.364   0.087   0.174   0.310   0.373   0.425   0.514 1.001
## theta[2]    0.387   0.082   0.213   0.337   0.392   0.443   0.536 1.001
## theta[3]    0.409   0.078   0.248   0.360   0.412   0.460   0.558 1.001
## theta[4]    0.409   0.078   0.244   0.361   0.412   0.461   0.559 1.001
## theta[5]    0.409   0.079   0.241   0.360   0.412   0.461   0.557 1.001
## theta[6]    0.409   0.080   0.243   0.360   0.412   0.462   0.561 1.001
## theta[7]    0.409   0.079   0.242   0.360   0.412   0.461   0.560 1.001
## theta[8]    0.408   0.079   0.242   0.359   0.411   0.460   0.559 1.001
## theta[9]    0.409   0.079   0.244   0.360   0.413   0.461   0.562 1.001
## theta[10]   0.410   0.079   0.243   0.361   0.412   0.462   0.559 1.001
## theta[11]   0.432   0.078   0.276   0.382   0.432   0.481   0.589 1.001
## theta[12]   0.433   0.078   0.276   0.383   0.433   0.482   0.588 1.001
## theta[13]   0.431   0.077   0.274   0.382   0.432   0.480   0.584 1.001
## theta[14]   0.431   0.078   0.274   0.382   0.432   0.481   0.586 1.001
## theta[15]   0.431   0.078   0.276   0.383   0.431   0.480   0.590 1.001
## theta[16]   0.454   0.078   0.306   0.403   0.451   0.503   0.619 1.001
## theta[17]   0.455   0.078   0.307   0.404   0.452   0.503   0.619 1.001
## theta[18]   0.454   0.079   0.302   0.403   0.451   0.503   0.618 1.001
## theta[19]   0.455   0.077   0.309   0.405   0.452   0.503   0.617 1.001
## theta[20]   0.454   0.079   0.302   0.402   0.451   0.503   0.619 1.001
## theta[21]   0.454   0.078   0.300   0.404   0.452   0.502   0.616 1.001
```

```
## theta[22]   0.455   0.079   0.303   0.404   0.452   0.503   0.622 1.001
## theta[23]   0.477   0.082   0.329   0.422   0.471   0.526   0.657 1.001
## theta[24]   0.477   0.081   0.329   0.423   0.472   0.526   0.655 1.001
## theta[25]   0.500   0.086   0.352   0.441   0.491   0.551   0.692 1.001
## theta[26]   0.499   0.085   0.353   0.441   0.491   0.549   0.690 1.001
## theta[27]   0.498   0.084   0.350   0.441   0.491   0.549   0.685 1.001
## theta[28]   0.522   0.091   0.368   0.458   0.512   0.577   0.725 1.001
## deviance  379.246   5.135 368.618 375.817 379.516 382.831 388.728 1.001
##           n.eff
## kappa     16000
## mu         8700
## theta[1]  18000
## theta[2]  20000
## theta[3]  17000
## theta[4]  15000
## theta[5]  19000
## theta[6]  20000
## theta[7]  12000
## theta[8]  11000
## theta[9]  20000
## theta[10] 20000
## theta[11] 13000
## theta[12]  9900
## theta[13] 12000
## theta[14] 20000
## theta[15] 20000
## theta[16] 13000
## theta[17] 20000
## theta[18] 18000
## theta[19] 20000
## theta[20] 15000
## theta[21]  6200
## theta[22] 17000
## theta[23] 20000
## theta[24] 20000
## theta[25] 20000
## theta[26] 20000
## theta[27] 20000
## theta[28] 12000
## deviance  20000
##
## For each parameter, n.eff is a crude measure of effective sample size,
## and Rhat is the potential scale reduction factor (at convergence, Rhat=1).
##
## DIC info (using the rule, pD = var(deviance)/2)
## pD = 13.2 and DIC = 392.4
## DIC is an estimate of expected predictive error (lower deviance is better).
```

```
TouchData0 <- list(
  Ntotal = 0,
  Nsubj = length(unique(TherapeuticTouch$s)),
  # y = TherapeuticTouch$y,
  # must convert subjects to sequence 1:Nsubj
  s = as.numeric(factor(TherapeuticTouch$s))
```

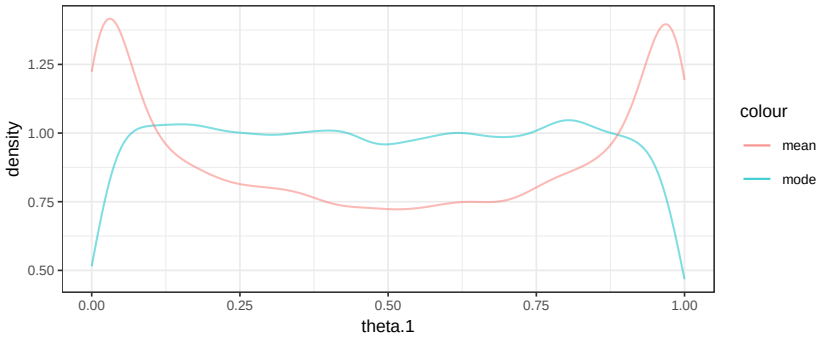
```
)
```

So why do we prefer the mode to the mean? Let's take a look at the prior distribution on one of the θs.

```
touch_jags_prior <-
  jags.parallel(
  data = TouchData0,
  model = touch_model,
  parameters.to.save = c("theta", "kappa", "omega"),
  n.burnin = 1000,
  n.iter = 41000,
  n.chains = 5,
  n.thin = 10,
  DIC = FALSE,
  jags.seed = 54321
)
```

```
touch_jags_prior3 <-
  jags.parallel(
  data = TouchData0,
  model = touch_model3,
  parameters.to.save = c("theta", "kappa", "mu"),
  n.burnin = 1000,
  n.iter = 41000,
  n.chains = 5,
  n.thin = 10,
  DIC = FALSE,
  jags.seed = 54321
)
```

```
gf_dens( ~ theta.1, data = posterior(touch_jags_prior), color = ~"mode") %>%
gf_dens( ~ theta.1, data = posterior(touch_jags_prior3), color = ~"mean")
```



Using the mean rather than mode corresponds to an unfortunate prior on θ_i.

9.7 Shrinkage

9.8 Example: Baseball Batting Average

9.9 Exerciess

Chapter 10

(Model Comparison)

Chapter 11

(NHST)

Chapter 12

(Point Null Hypotheses)

Chapter 13

(Goals, Power, Sample Size)

Chapter 14

Stan

14.1 Why Stan might work better

Stan is sometimes (but not always) better or faster than JAGS. The reason is that the HMC (Hamilton Markov Chain) algorithm that it uses avoids some of the potential problems of the Metropolis algorithm and Gibbs sampler. You can think of HMC as a generlization of the Metropolis algorithm. Recall that in the Metropolis algorithm

- There is always a current vector of parameter values (like the current island in our story)
- A new vector of parameter values is proposed
- The proposal is accepted or rejected by comparing the ratio of the likelihoods of the current and proposal vectors.
  - It is important that we only need the ratio because the scaling constant would be prohibitively expensive to compute.

The main change is in how HMC chooses its proposals. Recall that in the basic Metropolis algorithm

- the proposal distribution is symmetric
- the proposal distribution is the same no matter where the current parameter vector is.

This has some potential negative consequences

- When the current position is a region of relatively low posterior density, the algorithm is as likely to propose moves that go farther from the mode as toward it. This can be inefficient.
- The behavior of the algorithm can be greatly affected by the "step size" (how likely the proposal is to be close to or far from the current position).

HMC addresses these by using a proposal distribution that * Changes depending on the current position * Is more likely to make proposals in the direction of the mode

Unlike Gibbs samplers, HMC is not guided by the fixed directions corresponding to letting only one parameter value change at a time. This makes it easier for HMC to navigate posteriors that have narrow "ridges" that don't follow one of these primary directions, so Stan is less disturbed by correlations in the posterior distribution than JAGS is.

The basic idea of the HMC sampler in Stan is to turn the log posterior upsided down so it is bowl-shaped with its mode at the "bottom" and to imagine a small particle sliding along this surface after receiving a "flick" to get it moving. If the flick is in the direction of the mode, the particle will move farther. If it is away from the mode, it may go up hill for a while and then turn around. (The same thing may happen if it travels in the direction of the mode and overshoots.) If it is in some other direction, it will take a curved path that bends toward the mode. A proposal is generated by specifying * A direction and "force" for the flick (momentum) * The amount of "time" to let the particle move. At the end of the specified amount of

time, the particle will be at the proposal position. * A level of discretization used to simulate the motion of the particle.

In principle (ie, physics), every proposal can be excepted (as in the Gibbs sampler). In practice, because the simulated movement is discretized into a sequence of small segments, a rule is used that involves both the ratio of the posterior values and the ratio of the momentums of the current and proposal values. If the motion is simulated with many small segments, the proposal will nearly always be accepted, but it will take longer to do the simulation. On the other hand, if a cruder approximation is used, the simulation is faster, but the proposal is more likely to be rejected. "Time" is represented by the product of the number of steps used and the size of the steps: `steps * eps` (eps is short for espilon, but "steps time eps" has a aring to it). The step size (`eps`) is a tuning parameter of the algorithm, and things seem to work most efficiently if roughly 2/3 of proposals are accepted. The value of `eps` can be adjusted to attain something close to this goal.

Stan adds an extra bit to this algorithm. To avoid the inefficiency of overshooting and turning around, it tries to estimate when this will happen. The result is its No U-Turn Sampler (NUTS). There are a number of other features that lead to the complexity of Stan including

- Symbolic differentiation to determine the gradient of the posterior and momentum.
- Simulation techniques for the physics to minimize the inaccuracy created because of discretization.
- Techniques for dealing with parameters with bounded support.
- An inital phase that helps set the tuning parameters: step size (`eps`), time (`steps * eps`), and the distribution from which to sample the initial momentum.

Because of all these techinical details, it is easy to see why Stan may be much slower than JAGS in situations where JAGS works well. The flip-side is that Stan make work in situations where JAGS fails altogether or takes too much time to be of practical use. Generally, as models become more complex, Stan gains the advantage. For simple models, JAGS is often faster.

14.2 Describing a model to Stan

Coding Stan models is also a bit more complicated, but what we have learned about JAGS is helpful. RStudio also offers excellent support for Stan, so we won't have to use tricks like writing a "function" in R that isn't really R code to describe a JAGS model.

To use Stan in R we will load the `rstan` package and take advantage of RStudio's Stan chunks.

```
library(rstan)
rstan_options(auto_write = TRUE)  # saves some compiling
```

Stan descriptions have several sections (not all of which are required):

- data – declarations of variables to hold the data
- transformed data – transformations of data
- parameters – declaration of parameters
- transformed parameters – transformations of parameters
- model – description of prior and likelihood
- generated quantities – used to keep track of additional values Stan can compute at each step.

Here is an example of a simple Stan model:

```
data {
  int<lower=0> N;  // N is a non-negative integer
  int y[N];          // y is a length-N vector of integers
}
parameters {
  real<lower=0,upper=1> theta;  // theta is between 0 and 1
}
```

```
model {
  theta ~ beta (1,1);
  y ~ bernoulli(theta);
}
```

See if you can figure out what this model is doing.

You will also see that Stan requires some extra stuff compared to JAGS. In particular, we need to tell Stan which quantities are integers and which are reals, and also if there is an restriction to their domain.

Note: running the chunk above takes a little while. This is when Stan compiles the C code for the model and also works out the formulas for the gradient (derivatives). The result is a **dynamic shared object** (**DSO**).

To use this model in RStudio, put the code in a Stan chunk (one of the options from the insert menu) and set the `output.var` to the R variable that will store the results. In this case, we have named it `simple_stan` using the argument `output.var = "simple_stan"`. Behind the scenes, RStudio is calling `stan_code()` to pass information between R and Stan and to get Stan to do the compilation.

```
class(simple_stan)   # what kind of thing is this?
```

```
## [1] "stanmodel"
## attr(,"package")
## [1] "rstan"
```

```
simple_stan          # let's take a look
```

```
## S4 class stanmodel '83dbe9f99dbf55ff04494fdddf566a3d' coded as follows:
## data {
##   int<lower=0> N;  // N is a non-negative integer
##   int y[N];          // y is a length-N vector of integers
## }
## parameters {
##   real<lower=0,upper=1> theta;  // theta is between 0 and 1
## }
## model {
##   theta ~ beta (1,1);
##   y ~ bernoulli(theta);
## }
```

We still need to provide Stan some data and ask Stan to provide us with some posterior samples. We do this with the `sampling()` function. By separating this into a separate step, we can use the same compiled model with different data sets or different settings (more iterations, for example) without having to recompile.

```
simple_stanfit <-
  sampling(
    simple_stan,
    data  = list(
      N = 50,
      y = c(rep(1, 15), rep(0, 35))
    ),
    chains = 3,      # default is 4
    iter = 1000,     # default is 2000
    warmup = 200     # default is half of iter
  )
```

```
##
## SAMPLING FOR MODEL '83dbe9f99dbf55ff04494fdddf566a3d' NOW (CHAIN 1).
## Chain 1:
```

```
## Chain 1: Gradient evaluation took 1.8e-05 seconds
## Chain 1: 1000 transitions using 10 leapfrog steps per transition would take 0.18 seconds.
## Chain 1: Adjust your expectations accordingly!
## Chain 1:
## Chain 1:
## Chain 1: Iteration:   1 / 1000 [  0%]  (Warmup)
## Chain 1: Iteration: 100 / 1000 [ 10%]  (Warmup)
## Chain 1: Iteration: 200 / 1000 [ 20%]  (Warmup)
## Chain 1: Iteration: 201 / 1000 [ 20%]  (Sampling)
## Chain 1: Iteration: 300 / 1000 [ 30%]  (Sampling)
## Chain 1: Iteration: 400 / 1000 [ 40%]  (Sampling)
## Chain 1: Iteration: 500 / 1000 [ 50%]  (Sampling)
## Chain 1: Iteration: 600 / 1000 [ 60%]  (Sampling)
## Chain 1: Iteration: 700 / 1000 [ 70%]  (Sampling)
## Chain 1: Iteration: 800 / 1000 [ 80%]  (Sampling)
## Chain 1: Iteration: 900 / 1000 [ 90%]  (Sampling)
## Chain 1: Iteration: 1000 / 1000 [100%]  (Sampling)
## Chain 1:
## Chain 1:  Elapsed Time: 0.002587 seconds (Warm-up)
## Chain 1:                0.008582 seconds (Sampling)
## Chain 1:                0.011169 seconds (Total)
## Chain 1:
##
## SAMPLING FOR MODEL '83dbe9f99dbf55ff04494fdddf566a3d' 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 / 1000 [  0%]  (Warmup)
## Chain 2: Iteration: 100 / 1000 [ 10%]  (Warmup)
## Chain 2: Iteration: 200 / 1000 [ 20%]  (Warmup)
## Chain 2: Iteration: 201 / 1000 [ 20%]  (Sampling)
## Chain 2: Iteration: 300 / 1000 [ 30%]  (Sampling)
## Chain 2: Iteration: 400 / 1000 [ 40%]  (Sampling)
## Chain 2: Iteration: 500 / 1000 [ 50%]  (Sampling)
## Chain 2: Iteration: 600 / 1000 [ 60%]  (Sampling)
## Chain 2: Iteration: 700 / 1000 [ 70%]  (Sampling)
## Chain 2: Iteration: 800 / 1000 [ 80%]  (Sampling)
## Chain 2: Iteration: 900 / 1000 [ 90%]  (Sampling)
## Chain 2: Iteration: 1000 / 1000 [100%]  (Sampling)
## Chain 2:
## Chain 2:  Elapsed Time: 0.00285 seconds (Warm-up)
## Chain 2:                0.009552 seconds (Sampling)
## Chain 2:                0.012402 seconds (Total)
## Chain 2:
##
## SAMPLING FOR MODEL '83dbe9f99dbf55ff04494fdddf566a3d' NOW (CHAIN 3).
## Chain 3:
## Chain 3: Gradient evaluation took 6e-06 seconds
## Chain 3: 1000 transitions using 10 leapfrog steps per transition would take 0.06 seconds.
## Chain 3: Adjust your expectations accordingly!
## Chain 3:
```

```
## Chain 3:
## Chain 3: Iteration:   1 / 1000 [  0%]  (Warmup)
## Chain 3: Iteration: 100 / 1000 [ 10%]  (Warmup)
## Chain 3: Iteration: 200 / 1000 [ 20%]  (Warmup)
## Chain 3: Iteration: 201 / 1000 [ 20%]  (Sampling)
## Chain 3: Iteration: 300 / 1000 [ 30%]  (Sampling)
## Chain 3: Iteration: 400 / 1000 [ 40%]  (Sampling)
## Chain 3: Iteration: 500 / 1000 [ 50%]  (Sampling)
## Chain 3: Iteration: 600 / 1000 [ 60%]  (Sampling)
## Chain 3: Iteration: 700 / 1000 [ 70%]  (Sampling)
## Chain 3: Iteration: 800 / 1000 [ 80%]  (Sampling)
## Chain 3: Iteration: 900 / 1000 [ 90%]  (Sampling)
## Chain 3: Iteration: 1000 / 1000 [100%]  (Sampling)
## Chain 3:
## Chain 3:  Elapsed Time: 0.002747 seconds (Warm-up)
## Chain 3:                0.009355 seconds (Sampling)
## Chain 3:                0.012102 seconds (Total)
## Chain 3:
```

The output below looks similar to what we have seen from JAGS.

```
simple_stanfit
```

```
## Inference for Stan model: 83dbe9f99dbf55ff04494fdddf566a3d.
## 3 chains, each with iter=1000; warmup=200; thin=1;
## post-warmup draws per chain=800, total post-warmup draws=2400.
##
##         mean se_mean   sd   2.5%    25%    50%    75%  97.5% n_eff Rhat
## theta   0.30    0.00 0.06   0.19   0.26   0.30   0.35   0.44   961    1
## lp__  -32.63    0.02 0.75 -34.85 -32.81 -32.35 -32.15 -32.10  1106    1
##
## Samples were drawn using NUTS(diag_e) at Wed Mar 27 22:17:30 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).
```

There are a number of functions that can extract information from stanfit objects.

```
methods(class = "stanfit")
```

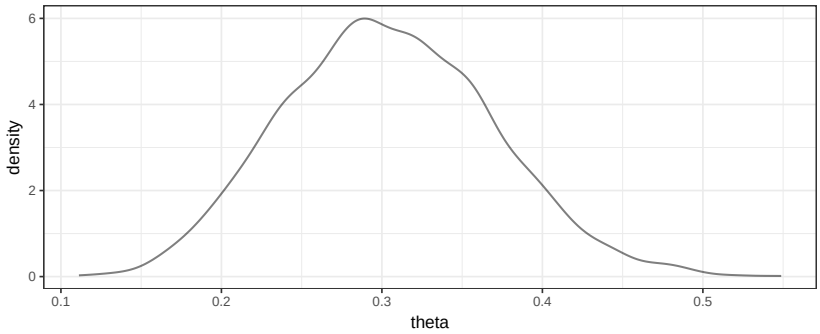
```
##  [1] as.array           as.data.frame      as.matrix
##  [4] as.mcmc.list       bridge_sampler     constrain_pars
##  [7] dim                dimnames           extract
## [10] get_cppo_mode      get_inits          get_logposterior
## [13] get_num_upars      get_posterior_mean get_seed
## [16] get_seeds          get_stancode       get_stanmodel
## [19] grad_log_prob      is.array           log_posterior
## [22] log_prob           loo                names
## [25] names<-            neff_ratio         nuts_params
## [28] pairs              plot               posterior
## [31] print              rhat               show
## [34] stanfit            summary            traceplot
## [37] unconstrain_pars
## see '?methods' for accessing help and source code
```

Unfortunately, some of these have the same names as functions elsewhere (in the coda package, for example). We generally adopt an approach that keeps things as similar to what we did with JAGS as possible.

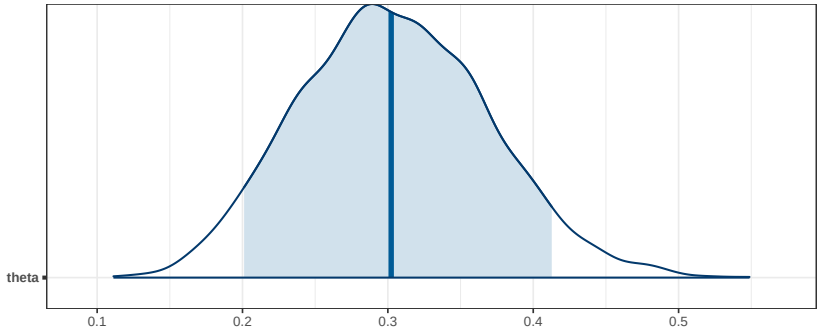
- Use `CalvinBayes::posterior()` to create a dataframe with posterior samples. These can be plotted or explored using `ggformula` or other familiar tools.

- Use `as.matrix()` or `as.mcmc.list()` to create an object that can be used with `bayesplot` just as we did when we used JAGS.

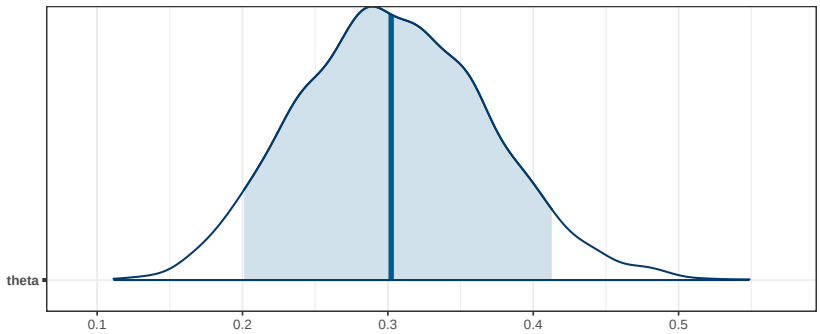
```
gf_dens(~theta, data = posterior(simple_stanfit))
```



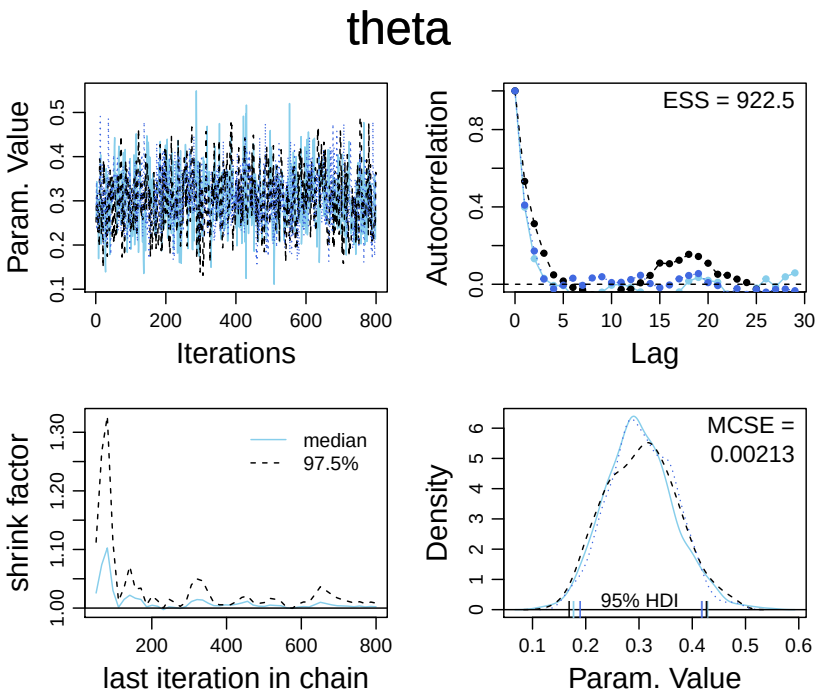

```
simple_mcmc <- as.matrix(simple_stanfit)
mcmc_areas(simple_mcmc, prob = 0.9, pars = "theta")
```




```
mcmc_areas(as.mcmc.list(simple_stanfit), prob = 0.9, pars = "theta")
```




```
diag_mcmc(as.mcmc.list(simple_stanfit))
```



14.3 Samping from the prior

In JAGS, to sample from the posterior, we just "removed the data". For any parameter values, the likelihood of not having any data if we don't collect any data is 1. So the posterior is the same as the prior.

Unlike JAGS, Stan does not allow missing data, so we need a different way to sample from the posterior. In Stan, we will remove the likelihood. To understand why this works, let's think a little bit about how Stan operates.

- All internal work is done on the log scale.

  log prior, log likelihood, log posterior.

- Additive constants on the log scale (multiplicative constants on the natural scale) don't matter…

  … at least not for generating posterior samples, so they can be ignored or chosen conveniently. The distribution functions in Stan are really logs of kernels with constants chosen to optimize efficiency of computation.

- log(posterior) = log(prior) + log(likelihood) + constant

  So if we don't add in the likelihood part, we just get the prior again as the posterior.

A line like

```
y ~ bernoulli(theta);
```

Is just telling stan to add the log of the bernoulli pmf for each value of the data vector `y` using the current value for `theta`. If we comment out that line, no log likelihood will be added.

```
data {
  int<lower=0> N;  // N is a non-negative integer
  int y[N];           // y is a length-N vector of integers
}
parameters {
  real<lower=0,upper=1> theta;  // theta is between 0 and 1
```

```
}
model {
  theta ~ beta (1,1);
  // y ~ bernoulli(theta);     // comment out to remove likelihood
}
```

```
simple0_stanfit <-
  sampling(
    simple0_stan,
    data  = list(
      N = 50,
      y = c(rep(1, 15), rep(0, 35))
    ),
    chains = 3,      # default is 4
    iter = 1000,     # default is 2000
    warmup = 200     # default is half of iter
  )
```

14.4 Exercises

1. Let's compare Stan and JAGS on the therapeutic touch example from Chapter 9. (See Figure 9.7 on page 236.) Stan and JAGS code for this example are below. The data are in `TherapeuticTouch`.

```
data {
  int<lower=1> Nsubj;
  int<lower=1> Ntotal;
  int<lower=0,upper=1> y[Ntotal];
  int<lower=1> s[Ntotal]; // notice Ntotal not Nsubj
}
parameters {
  real<lower=0,upper=1> theta[Nsubj]; // individual prob correct
  real<lower=0,upper=1> omega;        // group mode
  real<lower=0> kappaMinusTwo;        // group concentration minus two
}
transformed parameters {
  real<lower=0> kappa;
  kappa <- kappaMinusTwo + 2;
}
model {
  omega ~ beta(1, 1);
  kappaMinusTwo ~ gamma(1.105125, 0.1051249 );  // mode=1, sd=10
  theta ~ beta(omega * (kappa-2) + 1, (1 - omega) * (kappa-2) + 1);
  for ( i in 1:Ntotal ) {
    y[i] ~ bernoulli(theta[s[i]]);
  }
}
```

```
jags_model <- function() {
  for ( i in 1:Ntotal ) {
    y[i] ~ dbern( theta[s[i]] )
  }
  for (s in 1:Nsubj ) {
    theta[s] ~ dbeta(omega * (kappa-2) + 1, (1-omega) * (kappa-2) + 1)
  }
```

```
  omega ~ dbeta(1, 1)
  kappa <- kappaMinusTwo + 2
  kappaMinusTwo ~ dgamma(1.105125, 0.1051249)  # mode=1, sd=10
}
```

Now answer the following questions.

a. What does the transformed parameters block of the Stan code do?
b. In the Stan code, there are two lines with ~ in them. One is inside a for loop and the other not. Why?
c. Compile the Stan program, and note how long it takes.
d. Now generate posterior samples using both the JAGS and Stan versions. Do the produce the same posterior distribution? How do the effective sample sizes compare?
e. Tweak the settings until you get similar effective sample sizes and rhat values from both Stan and JAGS. (ESS is the best metric of how much work they have done, and we want to be sure both algorithms think they are converging to get a fair comparison.) Once you have done that, compare their speeds. Which is faster in this example? By how much? (If you want R to help automate the timing, you can use `system.time()`.)

Chapter 15

GLM Overview

15.1 Data consists of observations of variables

Rectangular format:

- rows: one per observation/observational unit (person, thing observed)
- columns: one per variable (measurement made/recorded)

15.1.1 Variable Roles

Often we will divid up variables into **predictor** or **explanatory** varaibles and **predicted** or **response** variables. This indicates that we want to use our model to help us predict the values of some variables given the values of other variables.

Example: How does a college predict success based on high school GPA and SAT/ACT scores?

15.1.2 Types of Variables

scale	metric?	continuous/discrete
ratio	M	C
interval	M	C
count	M	D
ordinal	-	D
nominal	-	D

15.2 GLM Framework

First try: $y = h(x_1, x_2, \ldots, x_k)$

- but we don't expect the predictors to exactly determine the response, so this is doomed to fail.

Second try: $y \sim h(x_1, x_2, \ldots, x_k)$

- predictors determine a **distribution** for response.
- this is essientially what we will do, but we will refine things a bit
  - this isn't always the easiest way to think about things

  - it's awkward to have h return a distribution
  - some features of the distribution will be determined outside of h
  - so h will tell us something about the distribution, but maybe not everything

- we have already seen this in action: Alice and Bob shooting free throws (or Reginald and Tony throwing at a target).
  - predictor: the subject (Alice or Bob)
  - response: hit or miss
  - distribution: Bernoulli with θ determined by the subject.
  - $h(Alice) = \theta_1$; $h(Bob) = \theta_2$

Chapter 16

Estimating One and Two Means

16.1 Basic Model for Two Means

16.1.1 Data

Two variables

- metric response
- dichotomous explanatory

16.1.2 Model

The traditional starting point for modeling means is to assume that the each group is sampled from a normal distribution with unknown mean and a **common standard deviation**. (We'll see that is no harder to have different standard deviations.)

So for two groups our model has three parameters: two means (μ_1 and μ_2) and one standard deviation σ.

Of course, we also need priors for these parameters. A common prior for the means is a normal distribution. We will start with a uniform prior for the standard deviation, but discuss better alternatives shortly.

That gives the following template for our model:

$$\begin{aligned} Y_{i|g} &\sim \mathsf{Norm}(\mu_g, \sigma) \\ \mu_g &\sim \mathsf{Norm}(?, ?) \\ \sigma &\sim \mathsf{Unif}(?, ?) \end{aligned}$$

The questions marks will be filled in based on considerations of the scale (order of magnitude of the data) and the amount of regularizing we want to do.

16.2 An Old Sleep Study

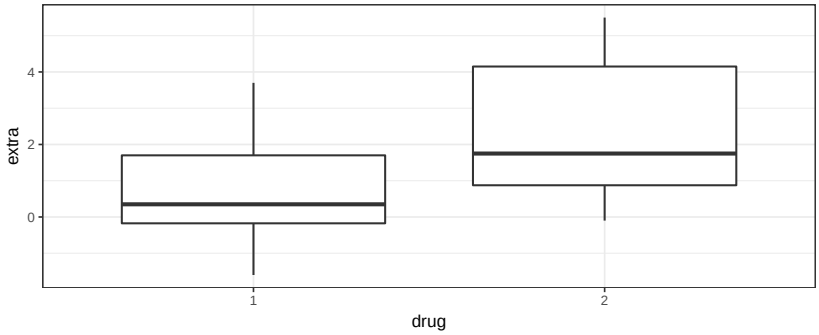
Cushny, A. R. and Peebles, A. R. (1905) "The action of optical isomers: II hyoscines." *The Journal of Physiology* 32, 501–510.

Design: Subjects sleep habits were compared without a sleep inducing drug and then with to see how two different drugs affected sleep.

16.2.1 Data

Let's look at the data. (`extra` = additional sleep on drug; `group` should really be `drug`, so let's rename it.)

```
library(ggformula)
library(dplyr)
sleep <-
  datasets::sleep %>% rename(drug = group)
gf_boxplot(extra ~ drug, data = sleep)
```




```
df_stats(extra ~ drug, data = sleep)
```

drug	min	Q1	median	Q3	max	mean	sd	n	missing
1	-1.6	-0.175	0.35	1.70	3.7	0.75	1.789	10	0
2	-0.1	0.875	1.75	4.15	5.5	2.33	2.002	10	0

16.2.2 Model

It is simple enough to convert the model description above into a JAGS model, but we need to fill in those question marks. Let's try this:

- mean for prior on μ_g: 0
  - corresponds to the drug having no impact on sleep
  - allows drug to increase or decrease sleep without prejudice
  - any other number would require more justification
  - will tend to pull estimates toward 0 (shrinkage) – we are requiring evidence to convince us that the drug does something to sleep.
- sd for prior on μ_g: 3
  - Says we are 95% certain that the average impact of a drug will be between -6 and 6 additional hours of sleep and that it is very unlikely the drug will change sleep by 9 or more hours. This is fairly week prior (6 extra hours of sleep would be a lot). This might be chosen in consultation with scientists who are more familiar with what is reasonable.
- range for σ: One crude way to set the prior is to give a ball mark estimate for the the standard deviation of the amount of sleep change in each treatment group and then make sure we cover a range 3 orders of magnitude in each direction.
- We can experiment with different priors to see how the impact results.

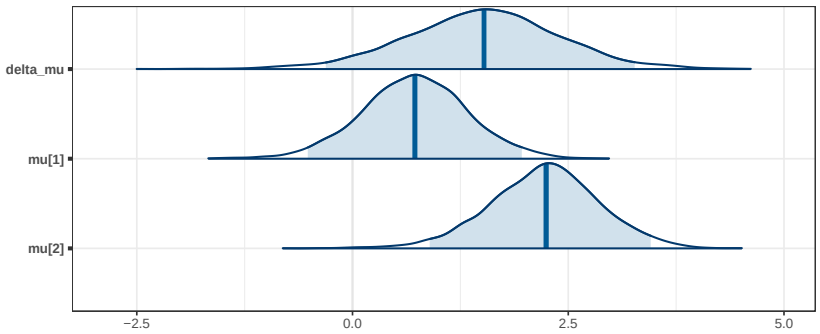
```
library(R2jags)
sleep_model <- function() {
  for (i in 1:Nobs) {
    extra[i] ~ dnorm(mu[drug[i]], 1 / sigma^2)
  }
  for (d in 1:Ndrugs) {
```

```
    mu[d] ~ dnorm(0, 1/3^2)           # sd = 3
  }
  sigma ~ dunif(2/1000, 2 * 1000)     # 3 orders of mag each way of 2
  delta_mu    <- mu[2] - mu[1]
  tau         <- 1 / sigma^2
}

sleep_jags <-
  jags(
    model = sleep_model,
    parameters.to.save = c("mu", "sigma", "delta_mu"),
    data = list(
      extra = sleep$extra,
      drug  = sleep$drug,
      Nobs  = nrow(sleep),
      Ndrugs = 2
    ),
    DIC = FALSE  # because we haven't discussed deviance yet
  )
```

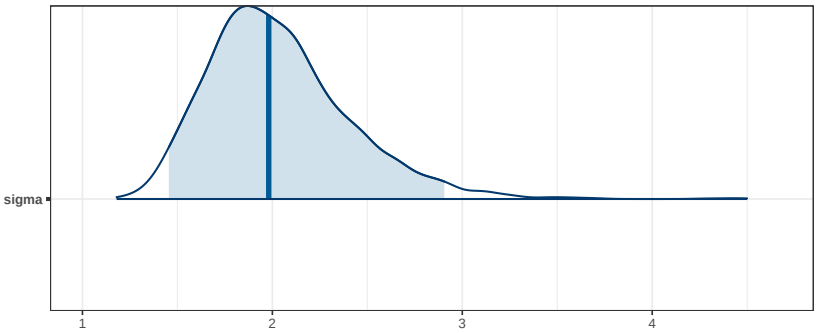
```
library(CalvinBayes)
library(bayesplot)
summary(sleep_jags)
```

```
## fit using jags
##  3 chains, each with 2000 iterations (first 1000 discarded)
##  n.sims = 3000 iterations saved
##          mu.vect sd.vect   2.5%   25%   50%   75% 97.5%  Rhat n.eff
## delta_mu   1.505   0.896 -0.309 0.933 1.523 2.075 3.269 1.001  3000
## mu[1]      0.717   0.622 -0.518 0.315 0.723 1.131 1.962 1.001  3000
## mu[2]      2.221   0.642  0.894 1.814 2.243 2.638 3.455 1.001  3000
## sigma      2.035   0.379  1.455 1.772 1.980 2.237 2.905 1.003   820
```

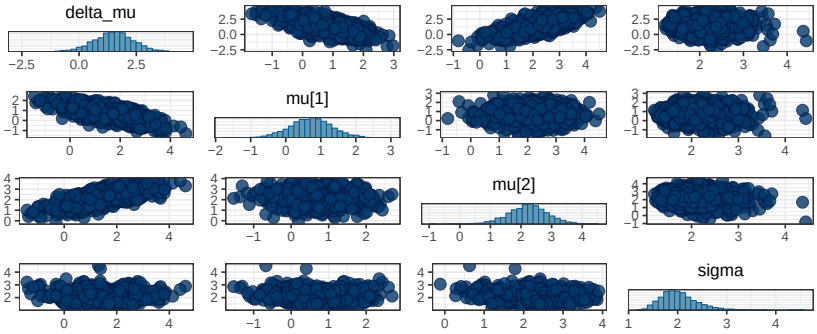
```
sleep_mcmc <- as.mcmc(sleep_jags)
mcmc_areas(sleep_mcmc, prob = 0.95, regex_pars = "mu")
```




```
mcmc_areas(sleep_mcmc, prob = 0.95, regex_pars = "sigma")
```




```
mcmc_pairs(sleep_mcmc)
```




```
mosaic::prop( ~(delta_mu > 0), data = posterior(sleep_jags))
```

```
## prop_TRUE
##     0.952
```

16.2.3 Separate standard deviations for each group

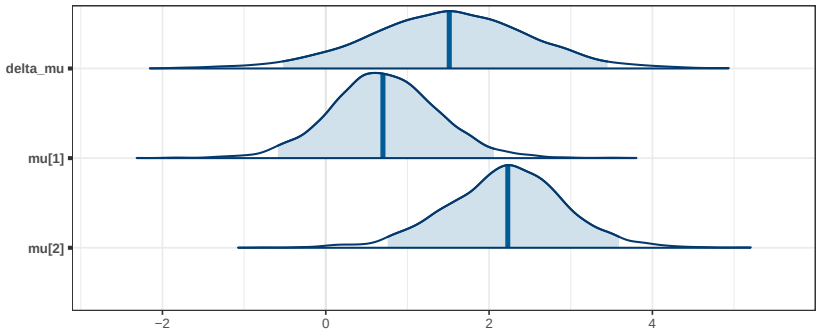
```
sleep_model2 <- function() {
  for (i in 1:Nobs) {
    extra[i] ~ dnorm(mu[drug[i]], 1/sigma[drug[i]]^2)
  }
  for (d in 1:Ndrugs) {
    mu[d]    ~  dnorm(0, 1/3^2)
    sigma[d] ~  dunif(2/1000, 2 * 1000)
    tau[d]   <- 1 / sigma[d]^2
  }
  delta_mu    <- mu[2] - mu[1]
  delta_sigma <- sigma[2] - sigma[1]
}

sleep_jags2 <-
  jags(
    model = sleep_model2,
    parameters.to.save = c("mu", "sigma", "delta_mu", "delta_sigma", "tau"),
    data = list(
      extra = sleep$extra,
      drug  = sleep$drug,
      Nobs  = nrow(sleep),
      Ndrugs = 2
    ),
```

```
    DIC = FALSE
  )
```

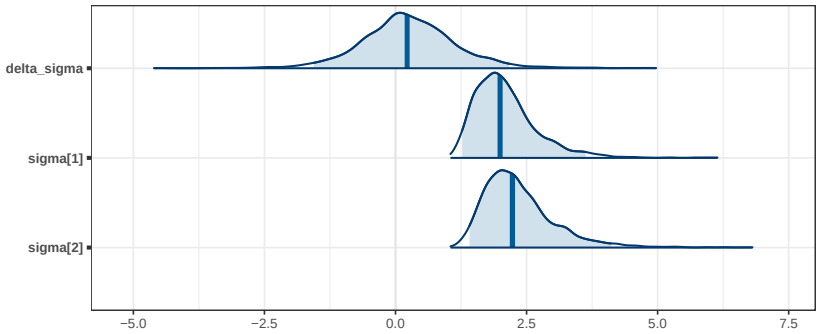
```
library(bayesplot)
library(CalvinBayes)
summary(sleep_jags2)
```

```
## fit using jags
##  3 chains, each with 2000 iterations (first 1000 discarded)
##  n.sims = 3000 iterations saved
##             mu.vect sd.vect   2.5%    25%   50%   75% 97.5%  Rhat n.eff
## delta_mu      1.491   1.002 -0.523  0.853 1.513 2.140 3.451 1.001  3000
## delta_sigma   0.254   0.935 -1.562 -0.298 0.223 0.783 2.170 1.001  3000
## mu[1]         0.712   0.672 -0.586  0.286 0.701 1.136 2.061 1.001  3000
## mu[2]         2.203   0.736  0.758  1.747 2.229 2.667 3.592 1.001  2600
## sigma[1]      2.103   0.611  1.272  1.679 1.994 2.373 3.627 1.001  3000
## sigma[2]      2.357   0.695  1.412  1.883 2.230 2.666 4.112 1.001  3000
## tau[1]        0.278   0.140  0.076  0.178 0.251 0.355 0.618 1.001  3000
## tau[2]        0.222   0.114  0.059  0.141 0.201 0.282 0.501 1.001  3000
```

```
sleep_mcmc2 <- as.mcmc(sleep_jags2)
mcmc_areas(sleep_mcmc2, prob = 0.95, regex_pars = "mu")
```




```
mcmc_areas(sleep_mcmc2, prob = 0.95, regex_pars = "sigma")
```




```
mosaic::prop( ~(delta_mu > 0), data = posterior(sleep_jags2))
```

```
## prop_TRUE
##    0.9303
```

```
mosaic::prop( ~(delta_sigma > 0), data = posterior(sleep_jags2))
```

```
## prop_TRUE
##      0.62
```

```
hdi(sleep_jags2, pars = c("delta"))
```

par	lo	hi	prob	chain

```
hdi(sleep_jags2)
```

par	lo	hi	prob	chain
delta_mu	-0.5231	3.2463	0.95	1
delta_mu	-0.4663	3.3997	0.95	2
delta_mu	-0.5670	3.6046	0.95	3
delta_sigma	-1.7227	2.1318	0.95	1
delta_sigma	-1.4971	2.1057	0.95	2
delta_sigma	-1.4255	2.2211	0.95	3
mu[1]	-0.5609	1.9123	0.95	1
mu[1]	-0.6925	1.9083	0.95	2
mu[1]	-0.5055	2.2318	0.95	3
mu[2]	0.7860	3.5920	0.95	1
mu[2]	0.7949	3.5497	0.95	2
mu[2]	0.6776	3.6541	0.95	3
sigma[1]	1.1026	3.3793	0.95	1
sigma[1]	1.1953	3.1850	0.95	2
sigma[1]	1.1495	3.3457	0.95	3
sigma[2]	1.2549	3.6323	0.95	1
sigma[2]	1.3421	3.7643	0.95	2
sigma[2]	1.2385	3.7057	0.95	3
tau[1]	0.0485	0.5655	0.95	1
tau[1]	0.0605	0.5789	0.95	2
tau[1]	0.0601	0.5433	0.95	3
tau[2]	0.0345	0.4542	0.95	1
tau[2]	0.0460	0.4268	0.95	2
tau[2]	0.0494	0.4590	0.95	3

16.2.4 Comparison to t-test

For those who know about 2-sample t tests:

```
t.test(extra ~ drug, data = sleep)
```

```
##
##  Welch Two Sample t-test
##
## data:  extra by drug
## t = -1.9, df = 18, p-value = 0.08
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -3.3655  0.2055
## sample estimates:
## mean in group 1 mean in group 2
##            0.75            2.33
```

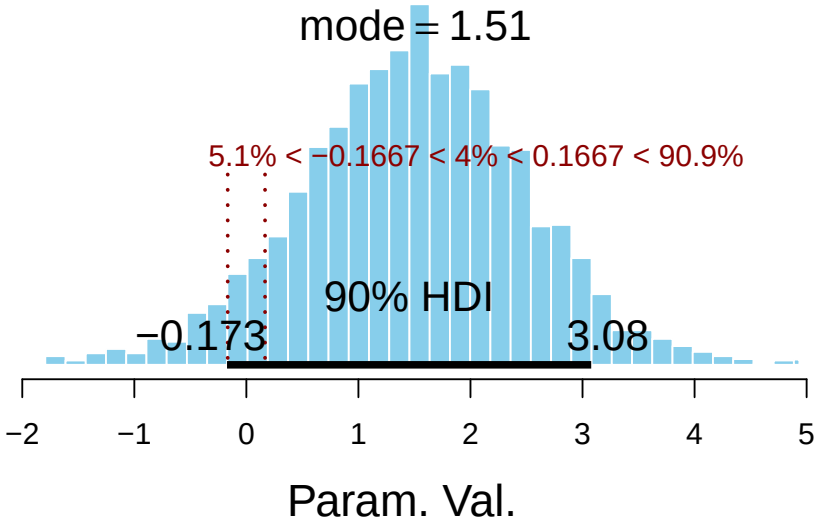
```
mosaic::prop( ~(delta_mu < 0), data = posterior(sleep_jags2))
```

```
## prop_TRUE
##   0.06967
```

16.2.5 ROPE (Region of Practical Equivalence)

Just knowing that two things are not the same is not of much practical use if the difference is small. One way to quantify this is to specify a **region of practical equivalence** (ROPE). We could decide, for example, that we are not interested in differences of less than 10 minutes (1/6 hours). Our ROPE (for the difference in means) would then be the interval $(-1/6, 1/6)$ and we could ask if there is evidence that the true difference lies outside that interval. This could be checked by seeing if an HDI lies completely outside the ROPE.

```
plot_post(posterior(sleep_jags2)$delta_mu, ROPE = c(-1/6, 1/6),
          hdi_prob = 0.9)
```




```
## $posterior
##       ESS  mean median mode
## var1 3000 1.491  1.513 1.51
##
## $hdi
##   prob      lo    hi
## 1  0.9 -0.1725 3.078
##
## $ROPE
##        lo     hi P(< ROPE) P(in ROPE) P(> ROPE)
## 1 -0.1667 0.1667   0.05067    0.04033     0.909
```

16.3 Variations on the theme

16.3.1 Other distributions for the response

While the normal distributions are commonly used to describe metric response variables, and many things are normally distributed, not everything is.

1. Skewed distributions.

If we have reason to believe that the shape of the response distribution (for a given combination of explanatory variables) is skewed (not symmetrical), then we have two options:

a. Transform the response variable.

Perhaps $log(y)$ or some other transformation of why is better described

```
by a normal distribution than $y$ itself.

b. Choose a skewed distribution.

Althernatively, we could choose a skewed distribution as part of the model.
```

2. Distributions with heavier tails.

Values that are quite far from the mean can have a large impact on what values of the man and standard deviation we find credible. If we suspect that the response distribution is likely to heavier tails, we can choose a family of distributions with heavier tails. This also makes our model more **robust against outliers** so that if the underlying distribution is normal but our data has an unusually large or small observation, the overall fit of the model is less disturbed.

The most commonly used family for this is the family of "student" t-distributions. If you have seen these before, you probably learned that

- the student t-distributions have one parameter, called **degrees of freedom** (usually denoted ν – that's the Greek letter ν). This is a shape parameter, and as $\nu \to \infty$, the t-distributions become more and more like the standard normal distribution. For this reason, we also may refer to ν as the **normality parameter**.
- the student t-distributions are **symmetric about 0**.
- the **mean**, **standard deviation**, **variance**, **precision** are given by

quantity	expression	valid when
mean	0	$\nu > 1$
standard deviation	$\sqrt{\frac{\nu}{\nu-2}}$	$\nu > 2$
variance	$\frac{\nu}{\nu-2}$	$\nu > 2$
standard deviation	$\frac{\nu-2}{\nu}$	$\nu > 2$

(These can fail to exist when the integral involved fail to converge.)

We can combine this information to form a more general family of t-distributions by shifting and scaling the student t-distributions. If $T \sim \mathsf{T}(\nu)$, then

$$T' = \mu + \frac{1}{\tau} T \sim \mathsf{T}(\mu, \tau, \nu)$$

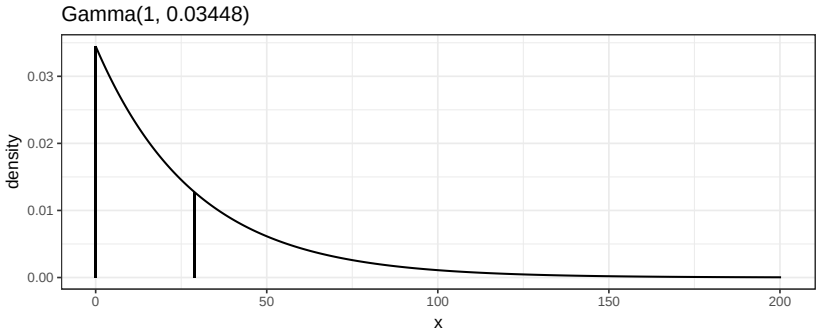
will have

- mean: μ (provided $\nu > 1$);
- standard deviation: $\sigma\sqrt{\frac{\nu}{\nu-2}}$ (provided $\nu > 2$) where $\sigma = \sqrt{\frac{1}{\tau}}$
- precision: $\tau\frac{\nu-2}{\nu}$

This is how JAGS defines the family of t-distributions. Note that τ is not exactly $\frac{1}{\sigma^2}$, but it is close when ν is large.

We can use this more general version of a t-distribution in place of a normal distribution by adding one additional prior – a prior for ν.
We want $\nu > 1$, so we will use our add and subtract trick to shift it to a distribution that is always positive. One choice would be a shifted exponential distribution (Gamma distribution with shape parameter 1). These distributions are heavily skewed, but this is appropriate since all the action is for small values of ν. If $\nu > 30$, the t distributions are hardly distinguishable from normal distributions. If we select a distribution with mean 30, then the distribution will be roughly evenly split between "basically normal" and "more spread out than a normal distribution".

```
# 29 because we will shift by 1
gamma_params(shape = 1, mean = 29, plot = TRUE)
```



shape	rate	scale	mode	mean	sd
1	0.0345	29	0	29	29

```
pexp(29, rate = 1/29)
```

```
## [1] 0.6321
```

```
sleep_model3 <- function() {
  for (i in 1:Nobs) {
    extra[i] ~ dt(mu[drug[i]], 1 / sigma[drug[i]]^2, nu[drug[i]])
  }
  for (d in 1:Ndrugs) {
    mu[d]          ~  dnorm(0, 1/3^2)
    sigma[d]       ~  dunif(2/1000, 2 * 1000)
    nuMinusOne[d] ~ dexp(1/29)
    nu[d]         <- nuMinusOne[d] + 1
  }
  delta_mu    <- mu[2] - mu[1]
  delta_sigma <- sigma[2] - sigma[1]
}

sleep_jags3 <-
  jags(
    model = sleep_model3,
    parameters.to.save = c("mu", "sigma", "delta_mu", "delta_sigma", "nu"),
    data = list(
      extra = sleep$extra,
      drug  = sleep$drug,
      Nobs  = nrow(sleep),
      Ndrugs = 2,
      n.iter = 5000
    ),
    DIC = FALSE
  )
```

```
## Warning in jags.model(model.file, data = data, inits = init.values,
## n.chains = n.chains, : Unused variable "n.iter" in data
```

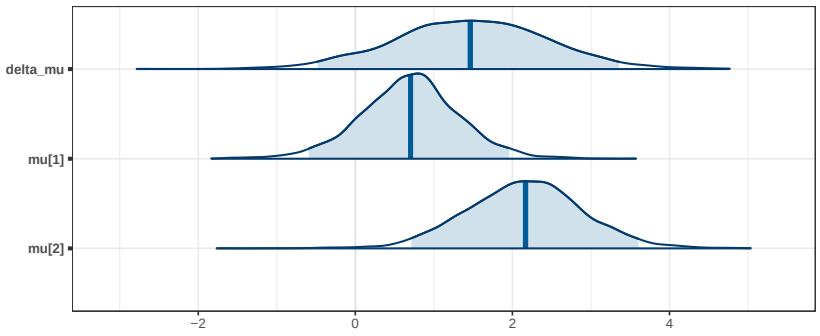
```
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 20
```

```
##    Unobserved stochastic nodes: 6
##    Total graph size: 67
##
## Initializing model
```

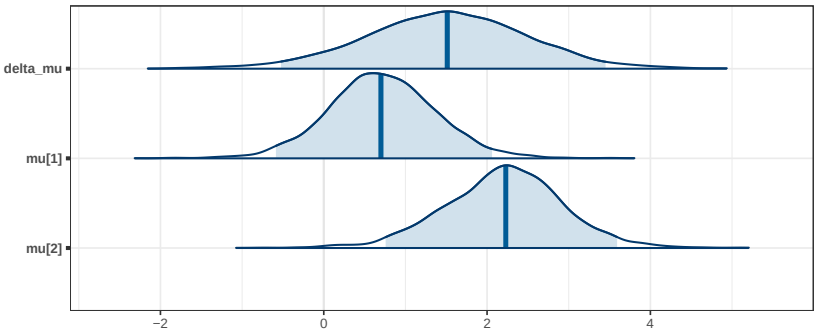
```
library(bayesplot)
library(CalvinBayes)
summary(sleep_jags3)
```

```
## fit using jags
##  3 chains, each with 2000 iterations (first 1000 discarded)
##  n.sims = 3000 iterations saved
##             mu.vect sd.vect   2.5%    25%    50%    75%   97.5%  Rhat
## delta_mu      1.464   0.992 -0.480  0.822  1.464  2.120   3.358 1.001
## delta_sigma   0.274   0.871 -1.400 -0.255  0.259  0.805   2.069 1.003
## mu[1]         0.691   0.641 -0.594  0.292  0.704  1.082   1.958 1.001
## mu[2]         2.155   0.751  0.714  1.666  2.167  2.631   3.606 1.001
## nu[1]        32.871  29.880  2.850 12.373 23.856 43.915 115.671 1.003
## nu[2]        33.855  29.672  3.043 12.434 25.551 46.011 111.424 1.001
## sigma[1]      2.009   0.585  1.150  1.592  1.908  2.330   3.349 1.002
## sigma[2]      2.282   0.666  1.308  1.831  2.156  2.604   3.985 1.002
##             n.eff
## delta_mu     3000
## delta_sigma  1000
## mu[1]        3000
## mu[2]        3000
## nu[1]         810
## nu[2]        3000
## sigma[1]     1500
## sigma[2]     2100
```

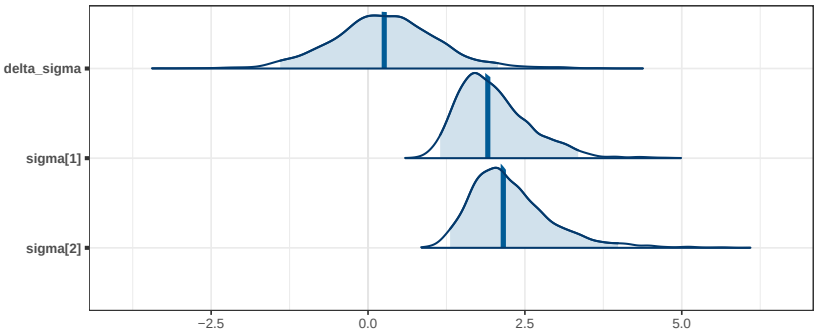
```
sleep_mcmc3 <- as.mcmc(sleep_jags3)
mcmc_areas(sleep_mcmc3, prob = 0.95, regex_pars = "mu")
```



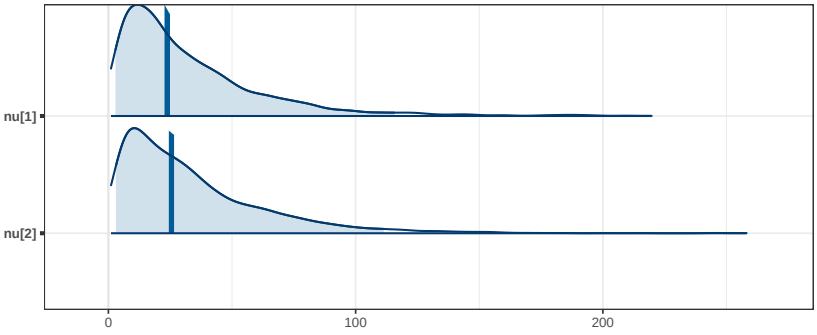

```
mcmc_areas(sleep_mcmc2, prob = 0.95, regex_pars = "mu")
```




```
mcmc_areas(sleep_mcmc3, prob = 0.95, regex_pars = "sigma")
```




```
mcmc_areas(sleep_mcmc3, prob = 0.95, regex_pars = "nu")
```




```
mosaic::prop( ~(delta_mu > 0), data = posterior(sleep_jags3))
```

```
## prop_TRUE
##    0.9257
```

```
hdi(sleep_jags3) %>% arrange(chain)
```

par	lo	hi	prob	chain
delta_mu	-0.5194	3.268	0.95	1
delta_sigma	-1.4178	1.780	0.95	1
mu[1]	-0.5883	1.883	0.95	1
mu[2]	0.8141	3.509	0.95	1
nu[1]	1.0686	96.255	0.95	1
nu[2]	1.0618	102.388	0.95	1
sigma[1]	1.0717	3.119	0.95	1
sigma[2]	1.2284	3.468	0.95	1
delta_mu	-0.3966	3.281	0.95	2
delta_sigma	-1.4576	2.037	0.95	2
mu[1]	-0.5978	2.056	0.95	2
mu[2]	0.7858	3.641	0.95	2
nu[1]	1.0433	89.376	0.95	2
nu[2]	1.0743	87.933	0.95	2
sigma[1]	1.0600	3.280	0.95	2
sigma[2]	1.2132	3.553	0.95	2
delta_mu	-0.6552	3.348	0.95	3
delta_sigma	-1.5312	2.035	0.95	3
mu[1]	-0.4334	2.054	0.95	3
mu[2]	0.5713	3.646	0.95	3
nu[1]	1.1046	81.929	0.95	3
nu[2]	1.1033	87.768	0.95	3
sigma[1]	1.0988	3.240	0.95	3
sigma[2]	1.1617	3.901	0.95	3

In this model we see that protecting our selves with a t distribution rather than a normal distribution doesn't seem to affect our estimates of `delta` much at all.

```
bind_rows(
  hdi(sleep_jags3, pars = "delta") %>% mutate(model = 3),
  hdi(sleep_jags2, pars = "delta") %>% mutate(model = 2)
) %>% arrange (chain, model)
```

```
## Warning in bind_rows_(x, .id): Unequal factor levels: coercing to character
```

```
## Warning in bind_rows_(x, .id): binding character and factor vector,
## coercing into character vector
```

```
## Warning in bind_rows_(x, .id): binding character and factor vector,
## coercing into character vector
```

par	lo	hi	prob	chain	model

16.3.2 Other Priors for σ (or τ)

Andrew Gelman has an entire paper devoted to the topic of choosing priors for standard deviation in Bayesian models. We won't delve into all the details or reasons to prefer (or avoid) some priors. But we will mention some of the alternatives available and make a few comments.

0. $\mathsf{Unif}(a/1000, 1000a)$ for some suitable a based on what we know about the scale of the data.

   We have already seen this above. It relisted here for completeness.

   Notes from Gelman's paper:

- Gelman mentions using $\mathsf{Unif}(0, A)$, which is similar, but doesn't avoid the really small values near 0.
- Gelman claims that this prior often works reasonably well in practice and is at least a good starting point, especially when the number of groups is small, as it is in our example.

1. Improper Uniform prior – $\mathsf{Unif}(0, \infty)$

Imagine we wanted to have a uniform prior in the interval $(0, \infty)$. A little thought shows that there is no such pdf (how tall would it need to be?) Nevertheless, some models can be fit with an improper prior – a function that has infinite area beneath it and so cannot be a kernel of a distribution. In this case, we could choose a constant function. If when multiplied by the likelihood we end up with something is a kernel, then we end up with a legitimate posterior, even though we used an improper prior. But generally speaking, improper priors are not the way to go. They can cause trouble for numerical algorithms (if they allow it at all), and there are usually better choices for priors.[1]

2. A proper prior with support $(0, \infty)$.

   a. Gamma

   The only family we know with support $(0, \infty)$ is the Gamma family. But we won't typically use it for this purpose. (But see below for its connection to τ.)

   b. Half-Distributions

   If we take the positive half of any distribution that is symmetric about 0, we get a "half-distibution". Half-normal and half-t distributions are frequently used as priors for the standard deviation. In particular, a half t distrubtion with 1 degree of freedom is called a half-Cauchy distribution. This distribution is so braod that it has not mean, so it functions somewhat like a proper substitute for the improper uniform prior.

   JAGS is clever enough to create the half distribution for us if we code things as if we are using a prior for σ that is symmetric around 0 – no need to say "half".

3. Dealing with precision (τ) directly.

   Instead of coming up with a prior for σ, we could instead come up with a prior for the variance (σ^2) or the precision (τ) instead. A gamma distribuiton is frequently used here because it happens to also be a conjugate prior in many situations involving normal distributions. The only tricky part here is choosing parameters for the gamma prior that correspond to our intitution since we are less familiar with precision than with standard deviation.

   Side note: The reason JAGS uses precision for normal distributions rather than standard deviation or variance is because Gamma distributions are a conjugate prior for precision in simple models based on the normal distribution, and BUGS (which predated JAGS) took advantage of conjugate priors to make sampling more efficient.

16.3.3 Paired Comparisons

The data actually contain another variable: `ID`. As it turns out, the same ten people were tested with each drug. If we are primarily interested in comparing the two drugs, we might take the difference between the extra sleep with one drug and with the other drug **for each person**. This is referred to as a paired design.

A paired comparison of means is really just looking at one mean – the mean difference. We can do this a couple of different ways:

1. Compute the difference before giving data to JAGS
2. Build the differences into the JAGS code.

[1] A related issue is an improper posterior. Some combinations of prior and likelihood can lead to a posterior with infinite "area/volume".

We will use option 1 here and convert our data so that each row corresponds to one person and there are separate columns for the extra sleep produced by each drug. This is sometimes referred to as converting from **long format** (more rows, fewer columns) to **wide format** (fewer rows, more columns). The `tidyr::spread()` function is useful for this. (And `tidyr::gather()` can be used to convert in the opposite direction.)

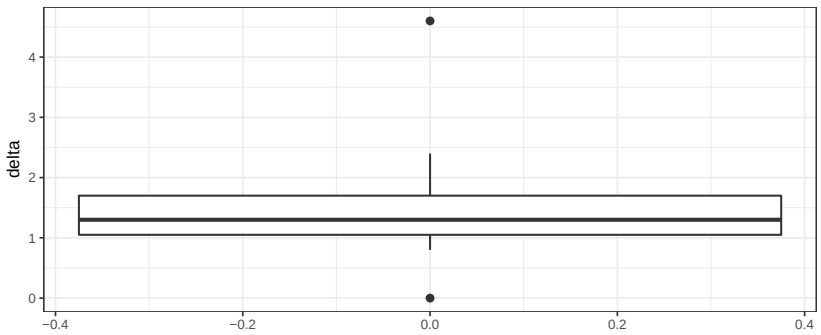
```
library(tidyr)
sleep_wide <-
  datasets::sleep %>%
  rename(drug = group) %>%
  mutate(drug = paste0("drug", drug)) %>%
  spread(key = drug, value = extra)
sleep_wide
```

ID	drug1	drug2
1	0.7	1.9
2	-1.6	0.8
3	-0.2	1.1
4	-1.2	0.1
5	-0.1	-0.1
6	3.4	4.4
7	3.7	5.5
8	0.8	1.6
9	0.0	4.6
10	2.0	3.4

```
sleep_wide <-
  sleep_wide %>%
  mutate(delta = drug2 - drug1)
sleep_wide
```

ID	drug1	drug2	delta
1	0.7	1.9	1.2
2	-1.6	0.8	2.4
3	-0.2	1.1	1.3
4	-1.2	0.1	1.3
5	-0.1	-0.1	0.0
6	3.4	4.4	1.0
7	3.7	5.5	1.8
8	0.8	1.6	0.8
9	0.0	4.6	4.6
10	2.0	3.4	1.4

```
gf_boxplot(~ delta, data = sleep_wide)
```



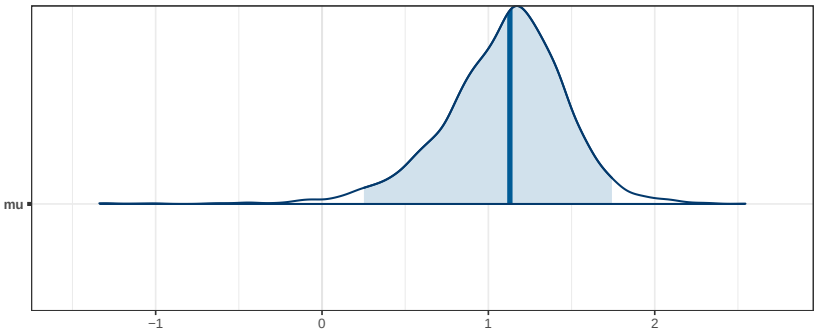
```
sleep_model4 <- function() {
  for (i in 1:Nsubj) {
    delta[i] ~ dt(mu, 1 / sigma^2, nu)
  }
  mu         ~ dnorm(0, 2)
  sigma      ~ dunif(2/1000, 2 * 1000)
  nuMinusOne ~ dexp(1/29)
  nu        <- nuMinusOne + 1
  tau       <- 1 / sigma^2
}

sleep_jags4 <-
  jags(
    model = sleep_model4,
    parameters.to.save = c("mu", "sigma", "nu"),
    data = list(
      delta = sleep_wide$delta,
      Nsubj = nrow(sleep_wide)
    ),
    n.iter = 5000,
    DIC = FALSE)
```

```
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 10
##    Unobserved stochastic nodes: 3
##    Total graph size: 25
##
## Initializing model
```

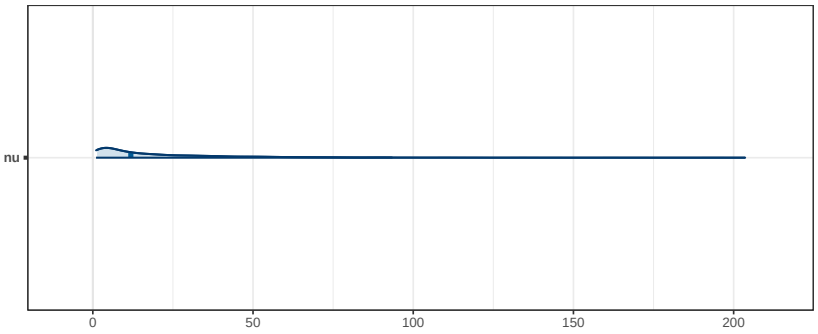
```
library(bayesplot)
library(CalvinBayes)
summary(sleep_jags4)
```

```
## fit using jags
##  3 chains, each with 5000 iterations (first 2500 discarded), n.thin = 2
##  n.sims = 3750 iterations saved
##       mu.vect sd.vect  2.5%   25%    50%    75%  97.5%  Rhat n.eff
## mu      1.092   0.379 0.252 0.886  1.129  1.334  1.743 1.002  3100
## nu     21.375  25.137 1.335 4.314 11.899 29.283 93.348 1.001  3100
## sigma   1.183   0.505 0.371 0.841  1.130  1.462  2.350 1.001  3800
```

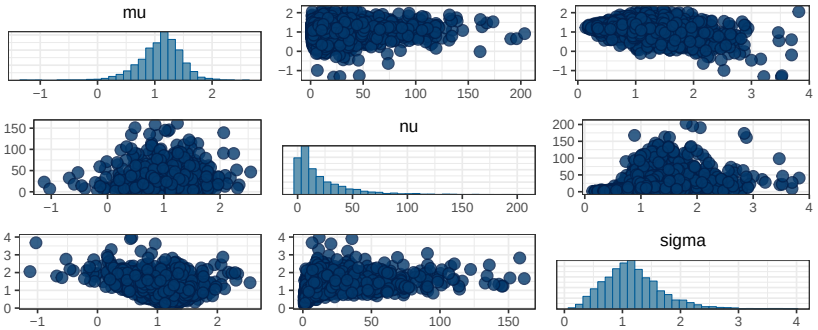
```
sleep_mcmc4 <- as.mcmc(sleep_jags4)
mcmc_areas(sleep_mcmc4, prob = 0.95, pars = "mu")
```




```
mcmc_areas(sleep_mcmc4, prob = 0.95, pars = "nu")
```




```
mcmc_pairs(sleep_mcmc4)
```




```
mosaic::prop( ~(mu > 0), data = posterior(sleep_jags4))
```

```
## prop_TRUE
##    0.9896
```

```
hdi(sleep_jags4, pars = c("mu"))
```

par	lo	hi	prob	chain
mu	0.2277	1.779	0.95	1
mu	0.3854	1.771	0.95	2
mu	0.2863	1.744	0.95	3

```
hdi(sleep_jags4)
```

par	lo	hi	prob	chain
mu	0.2277	1.779	0.95	1
mu	0.3854	1.771	0.95	2
mu	0.2863	1.744	0.95	3
nu	1.0013	73.264	0.95	1
nu	1.0067	65.680	0.95	2
nu	1.0194	77.905	0.95	3
sigma	0.2741	2.206	0.95	1
sigma	0.2526	2.074	0.95	2
sigma	0.3152	2.133	0.95	3

16.4 How many chains? How long?

16.4.1 Why multiple chains?

Multiple chains are primarily for diagnostics. Once we are sure things are behaving as they should, we could go back and run one really long chain if we wanted.

16.4.2 What large n.eff does and doesn't do for us

A large value of `n.eff` makes our estimates more stable. If we run them again, or compare multiple chains, the HDIs for parameters with larger `n.eff` will change the least. For important work, we will run with a modest `n.iter` until we are sure things are working well. Then we can increase the number of iterations for final analysis to make sure that our estimates are stable.

A large value of `n.eff` does not make our estimates "better" or make the posterior more concentrated.

16.5 Looking at Likelihood

```
likelihood <- function(mu1, sigma1, mu2 = mu1, sigma2 = sigma1, x, y = c(),
                       log = FALSE) {
  D <- tibble(
    group = c(rep("1", length(x)), rep("2", length(y))),
    l = c(
      dnorm(x, mu1, sigma1, log = log),
      dnorm(y, mu2, sigma2, log = log)),
    x = c(x, y)
  )
  if (log) {
    logL <- sum(D$l)
    L    <- exp(logL)

  } else {
    L    <- prod(D$l)
    logL <- log(L)
  }

  T <- tibble(x = mean(x), logL = logL, height = 1.2 * dnorm(0, 0, 1, log = log))

  cat(paste0("log likelihood: ", format(logL)),
```

```
      "; mu: ", format(mu1), ", ", format(mu2),
      "; sigma: ", format(sigma1), ", ", format(sigma2), "\n")

  gf_segment(0 + 1 ~ x + x, data = D, color = ~ group) %>%
    gf_point(0 ~ x, data = D, color = ~ group) %>%
    gf_function(function(x) dnorm(x, mu1, sigma1, log = log), color = ~"1") %>%
    gf_function(function(x) dnorm(x, mu2, sigma2, log = log), color = ~"2")
}
```

```
library(manipulate)
manipulate(
  likelihood(MU1, SIGMA1, MU2, SIGMA2,
             x = c(8, 12), y = c(11, 13), log = LOG) %>%
    gf_lims(x = c(0, 20), y = c(NA, 0.5)),
  MU1 = slider(5, 15, 10, step = 0.2),
  MU2 = slider(5, 15, 10, step = 0.2),
  SIGMA1 = slider(1, 10, 2, step = 0.2),
  SIGMA2 = slider(1, 10, 2, step = 0.2),
  LOG = checkbox(FALSE, "log likelihood")
)
```

16.6 Exercises

1. Using the 30-year-olds in the `NHANES` data set (in the NHANES package), fit a model that compares the mean height for men and women (allowing for different standard deviations). Then and answer the following questions about your model.

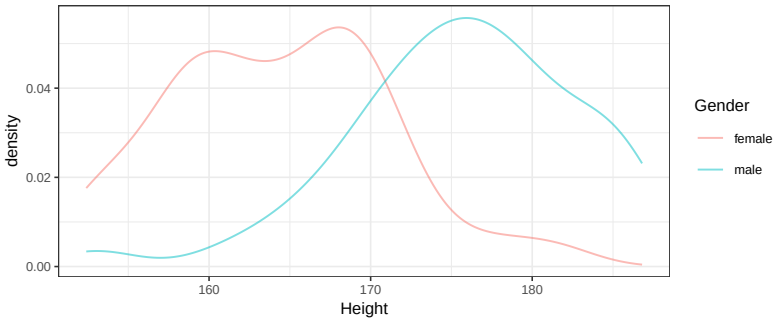
   a. Explain your choice of priors.
   b. Does this sample provide enough evidence to conclude that men are taller (on average)?
   c. Does this sample provide enough evidence to conclude that the standard deviation of height differs between men and women?

```
Thirty <- NHANES::NHANES %>% filter(Age == 30)
df_stats(Height ~ Gender, data = Thirty)
```

Gender	min	Q1	median	Q3	max	mean	sd	n	missing
female	152.4	159.0	163.9	168.6	181.3	164.3	6.731	73	3
male	153.0	171.7	176.2	181.1	186.8	175.7	7.082	90	0

```
gf_dens( ~ Height, color = ~ Gender, data = Thirty, binwidth = 1)
```

```
## Warning: Removed 3 rows containing non-finite values (stat_density).
```



2. Repeat exercise 1, but compare pulse instead of height.

3. The typical lifespan of a laboratory rat that eats ad lib is approximately 700 days. When rats are placed on a restricted diet, their longevity can increase, but there is a lot of variability in lifespans across different individual rats. Restricting the diet might not only affect the typical lifespan, but restricting the diet might also affect the variance of the lifespan across rats. We consider data from R. L. Berger, Boos, and Guess (1988), as reported in Hand, Daly, Lunn, McConway, and Ostrowski (1994, data set #242), and which are available as `CalvinBayes::RatLives`.

   a. Run the two-group analysis on the rat longevity data using a t distribution for the response variable. Do the groups appear to differ in their central tendencies and variances? Does the value of the normality parameter suggest that the distribution of lifetimes has heavier tails than a normal distribution?

   b. Did you plot the data before doing part a? It's a good idea to do that. Create a plot that compares the distributions of rat life for the two groups of rats in the data.

   c. Your plot should have revealed that within each group the data appear to be skewed to the left. That is, within each group, there are many rats that died relatively young, but there are fewer rats who lived especially long. We could try to implement a skewed noise distribution, or we could try to transform the data so they are approximately symmetric within each group after transformation. We will try the latter approach here. To get rid of leftward skew, we need a transformation that expands larger values more than the smaller values. We will try squaring the data. (You an use `mutate()` to add a new variable containing the square of the lifetimes of the rats to the original data or you can take care of this inside the list that you pass to JAGS). Do the groups appear to differ in their central tendencies and variances with this model? What does the value of the normality parameter suggest about the distribution of the transformed lifetimes?

   d. To compare the results of the two models, it is useful to back-transform to the natural scale. Give a 90% posterior HDI for the difference in mean lifetime based on each model. These should both be in units of days.

   e. Did you use ROPEs in any of your answers above? If not, go back and do so. (You will need to decide how wide the ROPE should be and if/how it changes when you apply the transformation.)

4. In the previous problem, how do the priors for the difference in mean lifetimes compare? Sample from the prior to find out. Be sure to deal appropriately with the transformation so that you are doing an apples to apples comparison.

5. Shohat-Ophir et al. (2012) were interested in alcohol preferences of sexually deprived male flies. The procedure is illustrated in Figure 16.13, and was described as follows:

   > One cohort, rejected-isolated, was subjected to courtship conditioning; they experienced 1-h sessions of sexual rejection by mated females, three times a day, for 4 days. …Flies in the mated-grouped cohort experienced 6-h sessions of mating with multiple receptive virgin females (ratio 1:5) for 4 days. Flies from each cohort were then tested in a two-choice preference assay, in which they voluntarily choose to consume food with or without 15% ethanol supplementation. (Shohat-Ophir et al., 2012, p. 1351, citations and figure reference removed)

   For each fly, the amount of each type of food consumed was converted to a preference ratio: the amount of ethanol-supplemented food minus the amount of regular food divided by the total of both. 3-day summary preference scores for each individual fruit fly were computed by summing the consumption of ethanol and non-ethanol across days 6–8. The amounts of food consumed and the preference ratios are in `CalvinBayes::ShohatOphirKAMH2012dataReduced`.

   a. How big are differences between groups relative to the uncertainty of the estimate? What do you conclude? (Answer this by computing what Kruschke calls the **effect size**. But note: effect size is not well defined; there are many things that go by that name. See, for example, the Wikipedia article on effect size.)

   b. Instead of focusing on the relative amounts of ethanol and regular food consumed, we might

also be interested in the absolute total amount of food consumed. Run the analysis on the total consumption data, which has column name `GrandTotal` in the data set. What do you conclude?

6. Redo problem 3 in Stan. You only need to do one model (transformed or untransformed, whichever works better).

   Note: The t distribution in Stan is parameterized differently. The normality paramter comes first, then mean, then standard deviation (not precision).

Chapter 17

Simple Linear Regression

Situation:

- Metric response
- Matric predictor

17.1 The deluxe basic model

17.1.1 Likelihood

$$y_i \sim \mathsf{Norm}(\mu_i, \sigma)$$
$$\mu_i \sim \beta_0 + \beta_1 x_i$$

Some variations:

- Replace normal distribution with something else (t is common).
- Allow standard deviations to vary with x as well as the mean.
- Use a different functional relationship between explanatory and response (non-linear regression)

Each of these is relatively easy to do. The first variation is sometimes called **robust regression** becuase it is more robust to unusual observations. Since it is no harder to work with t distributions than with normal distributions, that will become our go-to simple linear regression model.

$$y_i \sim \mathsf{T}(\mu_i, \sigma, \nu)$$
$$\mu_i \sim \beta_0 + \beta_1 x_i$$

17.1.2 Priors

We need priors for β_0, β_1, σ, and ν.

- ν: We've already seend that a **shifted Gamma** with mean around 30 works well as a generic prior giving the data room to stear us away from normality if warranted.
- β_1: The MLE for β_1 is

$$\hat{\beta}_1 = r\frac{SD_y}{SD_x}$$

so it makes sense to have a prior broadly covers the interval $(-\frac{SD_y}{SD_x}, \frac{SD_y}{SD_x})$.

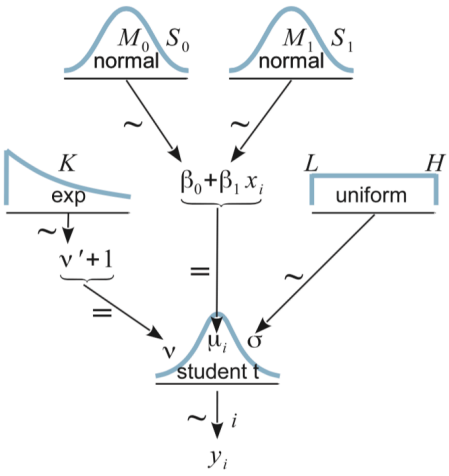
- β_0: The MLE for β_0 is

$$\hat{\beta}_0 = \overline{y} - \hat{\beta}_1 \overline{x} = \overline{y} - r\frac{SD_y}{SD_x} \cdot \overline{x}$$

so we can pick a prior that broadly covers the interval $(\overline{y} - \frac{SD_y}{SD_x} \cdot \overline{x}, \overline{y} - \frac{SD_y}{SD_x} \cdot \overline{x})$

- σ measures the amount of variability in responses for a *fixed value* of x (and is assumed to be the same for each x in the simple version of the model). A weakly informative prior should cover the range of reasonable values of σ with plenty of room to spare. (Our 2-or-3-orders-of-magnititude-either-way uniform distribution might be a reasonable starting point.)

Here's the big picture:



17.2 Example: Galton's Data

Since we are looking at regression, let's use an historical data set that was part of the origins of the regression story: Galton's data on height. Galton collected data on the heights of adults and their parents.

```
head(mosaicData::Galton)
```

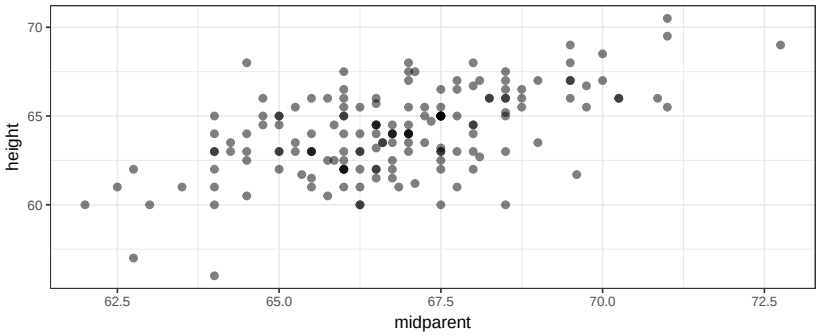
family	father	mother	sex	height	nkids
1	78.5	67.0	M	73.2	4
1	78.5	67.0	F	69.2	4
1	78.5	67.0	F	69.0	4
1	78.5	67.0	F	69.0	4
2	75.5	66.5	M	73.5	4
2	75.5	66.5	M	72.5	4

To keep things simpler for the moment, let's consider only women, and only one sibling per family.

```
set.seed(54321)
library(dplyr)
GaltonW <-
  mosaicData::Galton %>%
  filter(sex == "F") %>%
  group_by(family) %>%
  sample_n(1)
```

Galton was interested in how people's heights are related to their parents' heights. He compbined the parents' heights into the "mid-parent height", which was the average of the two.

```
GaltonW <-
  GaltonW %>%
  mutate(midparent = (father + mother) / 2)
gf_point(height ~ midparent, data = GaltonW, alpha = 0.5)
```



17.2.1 Describing the model to JAGS

```
galton_model <- function() {
  for (i in 1:length(y)) {
    y[i]   ~ dt(mu[i], 1/sigma^2, nu)
    mu[i] <- beta0 + beta1 * x[i]
  }
  sigma ~ dunif(6/100, 6 * 100)
  nuMinusOne ~ dexp(1/29)
  nu <- nuMinusOne + 1
  beta0 ~ dnorm(0, 1/100^2)    # 100 is order of magnitude of data
  beta1 ~ dnorm(0, 1/4^2)      # expect roughly 1-1 slope
}
```

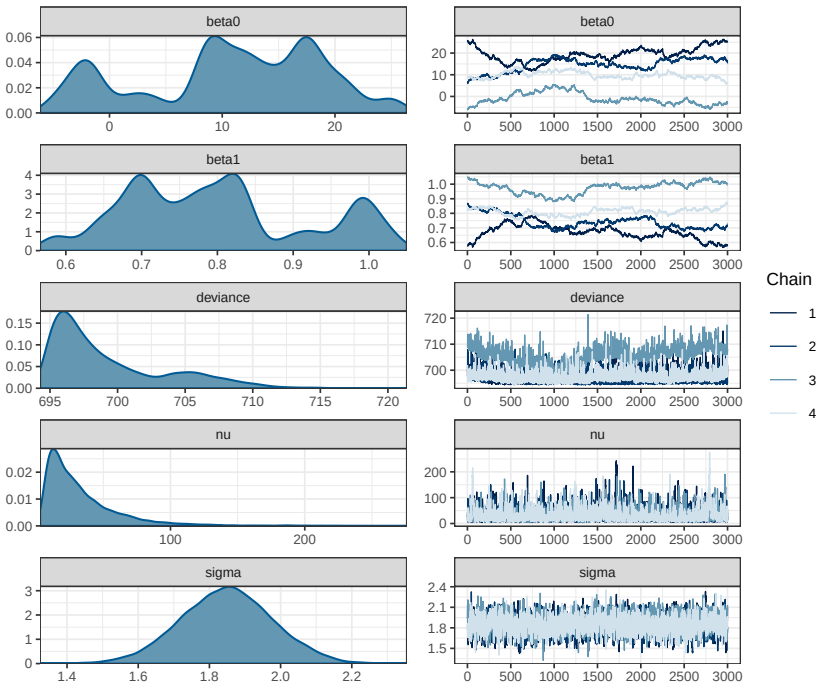
```
library(R2jags)
library(mosaic)
galton_jags <-
  jags(
    model = galton_model,
    data = list(y = GaltonW$height, x = GaltonW$midparent),
    parameters.to.save = c("beta0", "beta1", "sigma", "nu"),
    n.iter = 5000,
    n.burnin = 2000,
    n.chains = 4,
    n.thin = 1
  )
```

```
library(bayesplot)
library(CalvinBayes)
summary(galton_jags)
```

```
## fit using jags
##  4 chains, each with 5000 iterations (first 2000 discarded)
##  n.sims = 12000 iterations saved
##          mu.vect sd.vect    2.5%     25%     50%     75%   97.5%  Rhat n.eff
```

```
## beta0     10.695   8.008  -4.072   5.664  11.532  17.189  24.534 4.376     4
## beta1      0.800   0.120   0.593   0.702   0.787   0.877   1.020 4.071     4
## nu        33.016  27.311   6.134  14.178  24.807  42.854 106.351 1.002  1600
## sigma      1.849   0.130   1.594   1.761   1.849   1.935   2.102 1.020   140
## deviance 699.315   4.276 694.765 696.069 697.727 701.525 709.475 2.312     6
##
## DIC info (using the rule, pD = var(deviance)/2)
## pD = 3.3 and DIC = 702.6
```

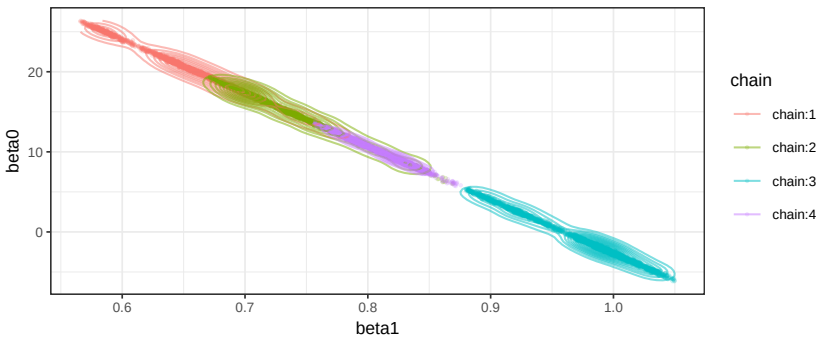
```
mcmc_combo(as.mcmc(galton_jags))
```



17.2.2 Problems and how to fix them

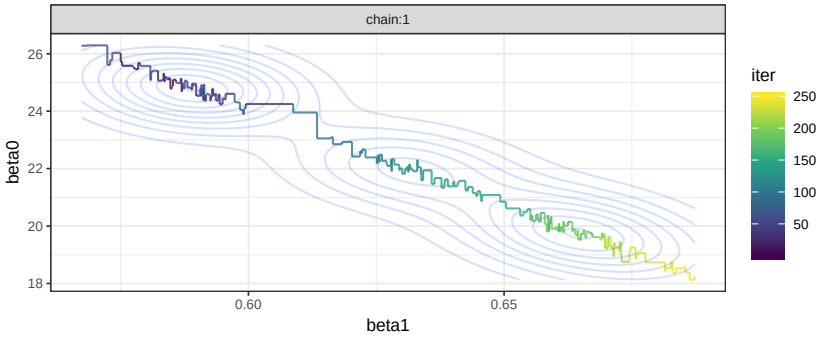
Clearly something is not working the way we would like with this model! Here's a clue as to the problem:

```
posterior(galton_jags) %>%
  gf_point(beta0 ~ beta1, color = ~ chain, alpha = 0.2, size = 0.4) %>%
  gf_density2d(alpha = 0.5)
```




```
posterior(galton_jags) %>% filter(iter <= 250, chain == "chain:1") %>%
  gf_step(beta0 ~ beta1, alpha = 0.8, color = ~iter) %>%
  gf_density2d(alpha = 0.2) %>%
```

```
  gf_refine(scale_color_viridis_c()) %>%
  gf_facet_wrap(~chain) #, scales = "free")
```



The correlation of the parameters in the posterior distribution produces a long, narrow, diagonal ridge that the Gibbs sampler samples only very slowly because it keeps bumping into edge of the cliff. (Remember, the Gibbs sampler only moves in "primary" directions.)

So how do we fix this? This is supposed to be the *simple* linear model after all. There are two ways we could hope to fix our problem.

1. **Reparameterize the model** so that the correlation between parameters (in the posterior distribution) is reduced or eliminated.
2. **Use a different algorithm** for posterior sampling. The problem is not with our model *per se*, rather it is with the method we are using (Gibbs) to sample from the posterior. Perhaps another algorithm will work better.

17.3 Centering and Standardizing

- Reparameterization 1: **centering**

  We can express this model as

$$
\begin{aligned}
y_i &\sim \mathsf{T}(\mu_i, \sigma, \nu) \\
\mu_i &= \alpha_0 + \alpha_1 (x_i - \overline{x})
\end{aligned}
$$

  Since

$$
\alpha_0 + \alpha_1 (x_i - \overline{x}) = (\alpha_0 - \alpha_1 \overline{x}) + \alpha_1 x_i
$$

  We see that $\beta_0 = \alpha_0 - \alpha_1 \overline{x}$ and $\beta_1 = \alpha_1$. So we can easily recover the original parameters if we like. (And if we are primarily interested in β_1, no translation is required.)

  This reparameterization maintains the natural scale of the data, and both α_0 and α_1 are easily interpreted: α_0 is the mean response when the predictor is the average of the predictor values *in the data*.

- Reparameterization 2: **standardization**

  We can also express our model as

$$z_{y_i} \sim \mathsf{T}(\mu_i, \sigma, \nu)$$

$$\mu_i = \alpha_0 + \alpha_1 z_{x_i}$$

$$z_{x_i} = \frac{x_i - \overline{x}}{SD_x}$$

$$z_{y_i} = \frac{y_i - \overline{y}}{SD_y}$$

Here the change in the model is due to a transformation of the data. Subtracting the mean and dividing by the standard deviation is called **standardization**, and the values produced are sometimes called **z-scores**. The resulting distributions of zy and zx will have mean 0 and standard deviation 1. So in addition to breaking the correlation pattern, we have now put things on a standard scale, regardless of what the original units were. This can be useful for picking constants in priors (we won't have to estimate the scale of the data involved). In addition, some algorithms work better if all the variables involved have roughly the same scale.

The downside is that we usually need to convert back to the original scales of x and y in order to interpret the results. But this is only a matter of a little easy algebra:

$$\begin{aligned}
\hat{z}_{y_i} &= \alpha_0 + \alpha_1 z x_i \\
\frac{\hat{y}_i - \overline{y}}{SD_y} &= \alpha_0 + \alpha_1 \frac{x_i - \overline{x}}{SD_x} \\
\hat{y}_i &= \overline{y} + \alpha_0 SD_y + \alpha_1 SD_y \frac{x_i - \overline{x}}{SD_x} \\
\hat{y}_i &= \underbrace{\left[\overline{y} + \alpha_0 SD_y - \alpha_1 \frac{SD_y}{SD_x}\overline{x}\right]}_{\beta_0} + \underbrace{\left[\alpha_1 \frac{SD_y}{SD_x}\right]}_{\beta_1} x_i
\end{aligned}$$

Since Kruscske demonstrates standardization, we'll do centering here.

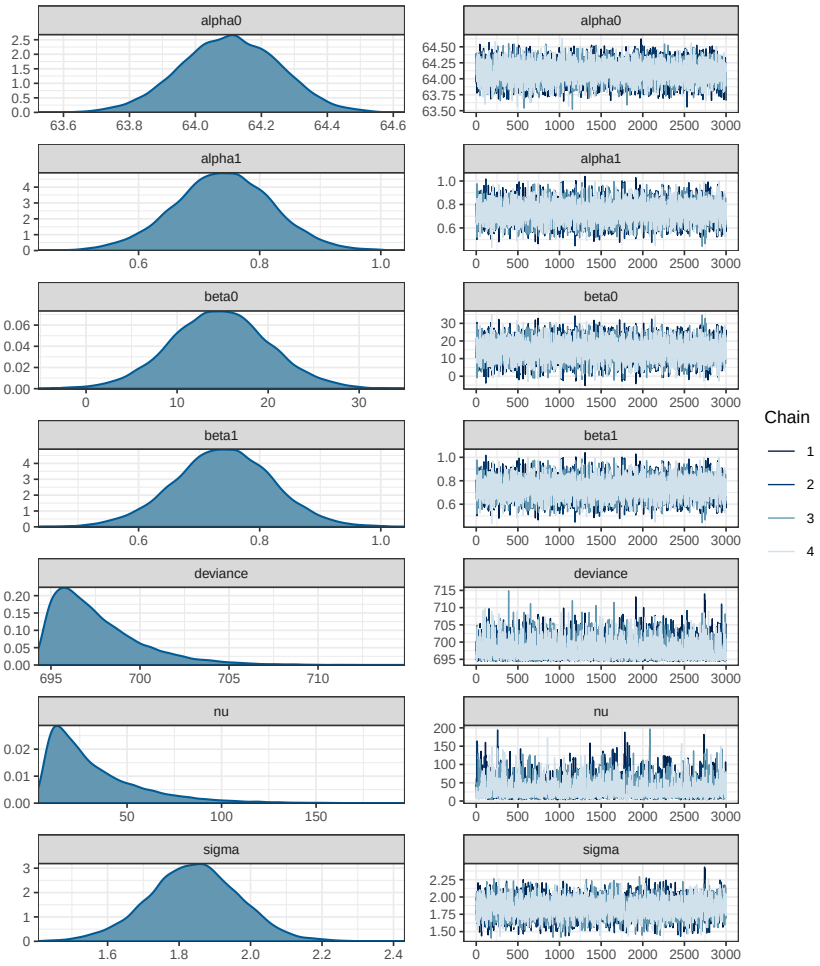
```
galtonC_model <- function() {
  for (i in 1:length(y)) {
    y[i]   ~ dt(mu[i], 1/sigma^2, nu)
    mu[i] <- alpha0 + alpha1 * (x[i] - mean(x))
  }
  sigma ~ dunif(6/100, 6 * 100)
  nuMinusOne ~ dexp(1/29)
  nu <- nuMinusOne + 1
  alpha0 ~ dnorm(0, 1/100^2)    # 100 is order of magnitude of data
  alpha1 ~ dnorm(0, 1/4^2)      # expect roughly 1-1 slope
  beta0 = alpha0 - alpha1 * mean(x)
  beta1 = alpha1                # not necessary, but gives us both names
}
galtonC_jags <-
  jags(
    model = galtonC_model,
```

```
    data = list(y = GaltonW$height, x = GaltonW$midparent),
    parameters.to.save = c("beta0", "beta1", "alpha0", "alpha1", "sigma", "nu"),
    n.iter = 5000,
    n.burnin = 2000,
    n.chains = 4,
    n.thin = 1
  )
```

```
summary(galtonC_jags)
```

```
## fit using jags
##  4 chains, each with 5000 iterations (first 2000 discarded)
##  n.sims = 12000 iterations saved
##          mu.vect sd.vect    2.5%     25%     50%     75%   97.5%  Rhat n.eff
## alpha0    64.105   0.149  63.812  64.004  64.106  64.208  64.390 1.001 12000
## alpha1     0.740   0.081   0.577   0.687   0.740   0.795   0.900 1.001 12000
## beta0     14.686   5.428   4.008  11.050  14.655  18.253  25.540 1.001 12000
## beta1      0.740   0.081   0.577   0.687   0.740   0.795   0.900 1.001 12000
## nu        32.250  24.769   6.281  14.368  24.592  42.701  98.596 1.001  8600
## sigma      1.841   0.128   1.585   1.757   1.842   1.926   2.091 1.001  5300
## deviance 697.587   2.496 694.651 695.740 696.961 698.787 704.017 1.001 12000
##
## DIC info (using the rule, pD = var(deviance)/2)
## pD = 3.1 and DIC = 700.7
```

```
mcmc_combo(as.mcmc(galtonC_jags))
```

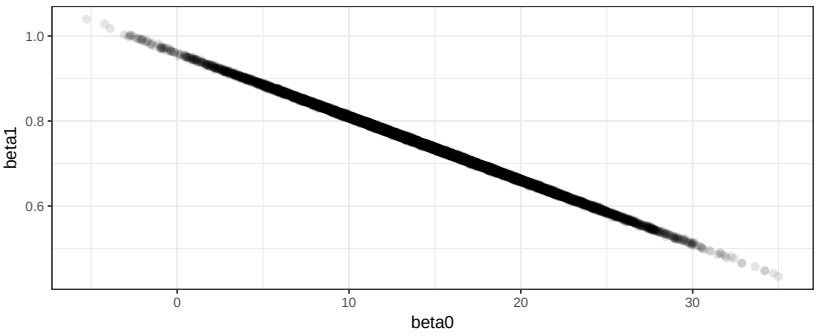


Ah! That looks much better than before.

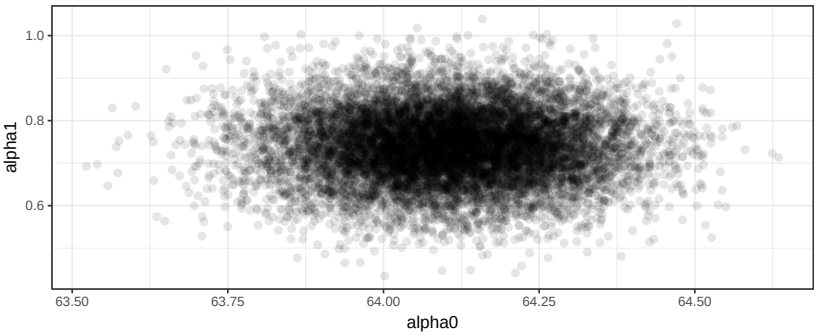
17.3.1 β_0 and β_1 are still correlated

Reparameterization has not changed our model, only the way it is described. In particular, β_0 and β_1 remain correlated in the posterior. But α_0 and α_1 are not correlated, and these are the parameters JAGS is using to sample.

```
gf_point(beta1 ~ beta0, data = posterior(galtonC_jags), alpha = 0.1)
```




```
gf_point(alpha1 ~ alpha0, data = posterior(galtonC_jags), alpha = 0.1)
```



17.4 We've fit a model, now what?

After centering or standardizing, JAGS works much better. We can now sample from our posterior distribution. But what do we do with our posterior samples?

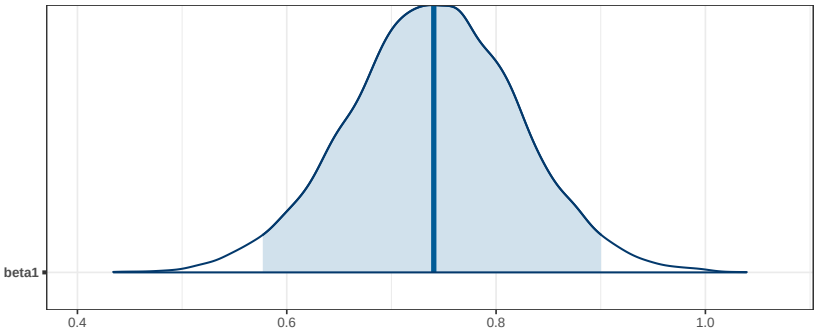
17.4.1 Estimate parameters

If we are primarily interested in a regression parameter (usually the slope parameter is much more interesting than the intercept parameter), we can use an HDI to express our estimate.

```
hdi(posterior(galtonC_jags), pars = "beta1")
```

par	lo	hi	prob
beta1	0.5825	0.9044	0.95

```
mcmc_areas(as.mcmc(galtonC_jags), pars = "beta1", prob = 0.95)
```



Galton noticed what we see here: that the slope is less than 1. This means that children of taller than average parents tend to be shorter than their parents and children of below average parents tend to be taller than their parents. He referred to this in his paper as "regression towards mediocrity". As it turns out, this was not a special feature of the heridity of heights but a general feature of linear models. Find out more in this Wikipedia artilce.

17.4.2 Make predictions

Suppse we know the heights of a father and mother, from which we compute ther mid-parent height x. How tall would we predict their daughters will be as adults? Each posterior sample provides an answer by describing a t distribution with `nu` degrees of freedom, mean $\beta_0 + \beta_1 x$, and standard deviation σ.

The posterior distribution of the average hieght of daughters born to parents with midparent height $x = 70$ is shown below, along with an HDI.

```
posterior(galtonC_jags) %>%
  mutate(mean_daughter = beta0 + beta1 * 70) %>%
  gf_dens(~mean_daughter)
```

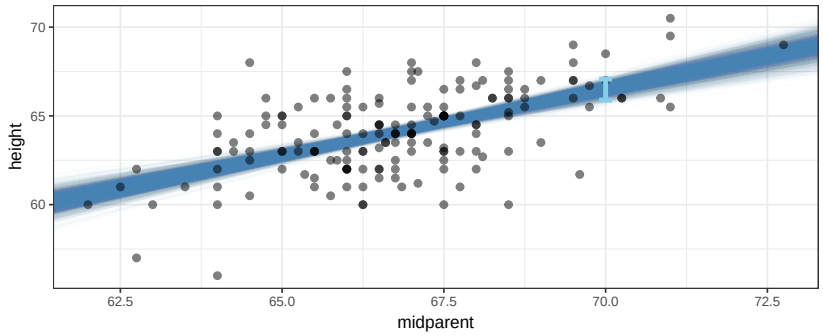

```
Galton_hdi <-
  posterior(galtonC_jags) %>%
  mutate(mean_daughter = beta0 + beta1 * 70) %>%
  hdi(pars = "mean_daughter")
Galton_hdi
```

par	lo	hi	prob
mean_daughter	65.89	67.07	0.95

So on average, we would predict the daughters to be about 66 or 67 inches tall.

We can visualize this by drawing a line for each posterior sample. The HDI should span the middle 95% of these.

```
gf_abline(intercept = ~beta0, slope = ~beta1, alpha = 0.01,
          color = "steelblue",
          data = posterior(galtonC_jags) %>% sample_n(2000)) %>%
  gf_point(height ~ midparent, data = GaltonW,
           inherit = FALSE, alpha = 0.5) %>%
  gf_errorbar(lo + hi ~ 70, data = Galton_hdi, color = "skyblue",
              width = 0.2, size = 1.2, inherit = FALSE)
```

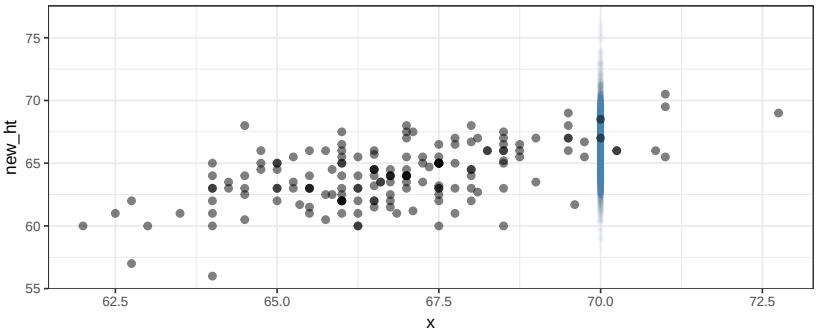


But this may not be the sort of prediction we want. Notice that most daughters' heights are not in the blue band in the picture. That band tells about the *mean* but doesn't take into account how much individuals vary about that mean. We can add that information in by taking our estimate for σ into account.

Here we generate heights by adding noise to the estimate given by values of β_0 and β_1.

```
posterior(galtonC_jags) %>%
  mutate(new_ht = beta0 + beta1 * 70 + rt(1200, df = nu) * sigma) %>%
  gf_point(new_ht ~ 70, alpha = 0.01, size = 0.7, color = "steelblue") %>%
```

```
  gf_point(height ~ midparent, data = GaltonW,
           inherit = FALSE, alpha = 0.5)
```




```
Galton_hdi2 <-
  posterior(galtonC_jags) %>%
  mutate(new_ht = beta0 + beta1 * 70 + rt(1200, df = nu) * sigma) %>%
  hdi(regex_pars = "new")
Galton_hdi2
```

par	lo	hi	prob
new_ht	62.38	70.53	0.95

So our model expects that most daughters whose parents have a midparent height of 70 inches are between 62.4 and 70.5 inches tall. Notice that this interval is taking into account both the uncertainty in our estimates of the parameters β_0, β_1, σ, and ν and the variability in heights that σ and ν indicate.

With a little more work, we can create intervals like this at several different midparent heights.

```
Post_galtonC <- posterior(galtonC_jags)

Grid <-
  expand.grid(midparent = 60:75, iter = 1:nrow(Post_galtonC))

posterior(galtonC_jags) %>%
  mutate(noise = rt(12000, df = nu)) %>%
  left_join(Grid) %>%
  mutate(height = beta0 + beta1 * midparent + noise * sigma) %>%
  group_by(midparent) %>%
  do(hdi(., pars = "height"))
```

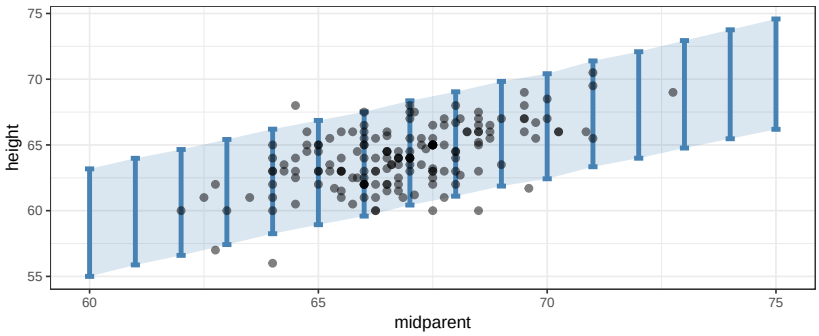
```
## Joining, by = "iter"
```

```
## # A tibble: 16 x 5
## # Groups:   midparent [16]
##    midparent par       lo    hi  prob
##        <int> <fct>  <dbl> <dbl> <dbl>
##  1        60 height  55.2  63.2  0.95
##  2        61 height  55.9  63.9  0.95
##  3        62 height  56.7  64.6  0.95
##  4        63 height  57.4  65.3  0.95
##  5        64 height  58.3  66.1  0.95
##  6        65 height  59.1  66.8  0.95
##  7        66 height  59.7  67.4  0.95
##  8        67 height  60.5  68.2  0.95
##  9        68 height  61.2  69.0  0.95
## 10        69 height  61.9  69.6  0.95
```

```
## 11         70 height  62.6  70.4  0.95
## 12         71 height  63.1  70.9  0.95
## 13         72 height  63.9  71.8  0.95
## 14         73 height  64.7  72.5  0.95
## 15         74 height  65.4  73.3  0.95
## 16         75 height  66.0  74.1  0.95
```

```
posterior(galtonC_jags) %>%
  mutate(noise = rt(12000, df = nu)) %>%
  left_join(Grid) %>%
  mutate(avg_height = beta0 + beta1 * midparent,
         height = avg_height + noise * sigma) %>%
  group_by(midparent) %>%
  do(hdi(., pars = "height")) %>%
  gf_ribbon(lo + hi ~ midparent, fill = "steelblue", alpha = 0.2) %>%
  gf_errorbar(lo + hi ~ midparent, width = 0.2, color = "steelblue", size = 1.2) %>%
  gf_point(height ~ midparent, data = GaltonW,
           inherit = FALSE, alpha = 0.5)
```

```
## Joining, by = "iter"
```



Comparing the data to the posterior predictions of the model is called a **posterior predictive check**; we are checking to see whether the data are consistent with what our posterior distribution would predict. In this case, things look good: most, but not all of the data is falling inside the band where our models predicts 95% of new observations would fall.

If the posterior predictive check indicates systematic problems with our model, it may lead us to propose another (we hope better) model.

17.5 Fitting models with Stan

Centering (or standardizing) is sufficient to make JAGS efficient enough to use. But we can also use Stan, and since Stan is not bothered by correlation in the posterior the way JAGS is, Stan works well even without reparamterizing the model.

Here is the Stan equivalent to our original JAGS model.

```
data {
  int<lower=0> N;     // N is a non-negative integer
  real y[N];          // y is a length-N vector of reals
  real x[N];          // x is a length-N vector of reals
}
parameters {
  real beta0;
  real beta1;
  real<lower=0> sigma;
  real<lower=0> nuMinusOne;
}
transformed parameters{
  real<lower=0> nu;
  nu = nuMinusOne + 1;
}
model {
  for (i in 1:N) {
    y[i] ~ student_t(nu, beta0 + beta1 * x[i], sigma);
  }
  beta0 ~ normal(0, 100);
  beta1 ~ normal(0, 4);
  sigma ~ uniform(6.0 / 100.0, 6.0 * 100.0);
  nuMinusOne ~ exponential(1/29.0);
}
```

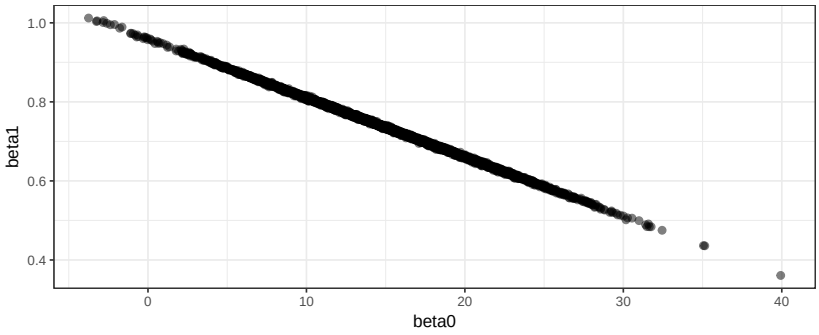
```
library(rstan)
galton_stanfit <-
  sampling(
    galton_stan,
    data = list(
      N = nrow(GaltonW),
      x = GaltonW$midparent,
      y = GaltonW$height
    ),
    chains = 4,
    iter = 2000,
    warmup = 1000
  )
```

```
galton_stanfit
```

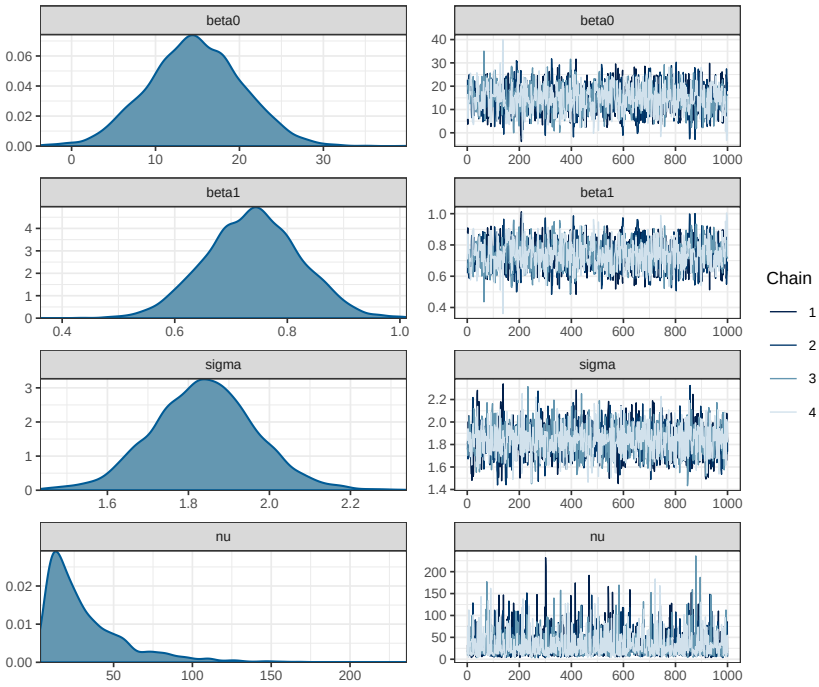
```
## Inference for Stan model: d35148cb031c8088132ce1e0ec66e1bb.
## 4 chains, each with iter=2000; warmup=1000; thin=1;
## post-warmup draws per chain=1000, total post-warmup draws=4000.
##
##               mean se_mean    sd    2.5%     25%     50%     75%   97.5% n_eff Rhat
## beta0        14.73    0.16  5.59    3.98   11.00   14.71   18.42   25.49  1254    1
## beta1         0.74    0.00  0.08    0.58    0.68    0.74    0.80    0.90  1252    1
```

```
## sigma          1.84    0.00  0.13    1.59    1.75    1.84    1.92    2.09  2018    1
## nuMinusOne    31.28    0.57 26.91    5.12   12.76   22.60   41.15  106.34  2230    1
## nu            32.28    0.57 26.91    6.12   13.76   23.60   42.15  107.34  2230    1
## lp__        -250.05    0.04  1.47 -253.84 -250.71 -249.70 -248.98 -248.30  1351    1
##
## Samples were drawn using NUTS(diag_e) at Wed Mar 27 22:20:46 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).
```

```
gf_point(beta1 ~ beta0, data = posterior(galton_stanfit), alpha = 0.5)
```




```
mcmc_combo(as.mcmc.list(galton_stanfit),
           pars = c("beta0", "beta1", "sigma", "nu"))
```



17.6 Exercises

1. Use Galton's data on the men to estimate

    a. The average of height of men whose parents are 65 and 72 inches tall.
    b. The middle 50% of heights of men whose parents are 65 and 72 inches tall.

You may use either JAGS or Stan.

2. When centering, why did we center x but not y?