(Re)Doing Bayesain Data Analysis

R Pruim

2019-02-04

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

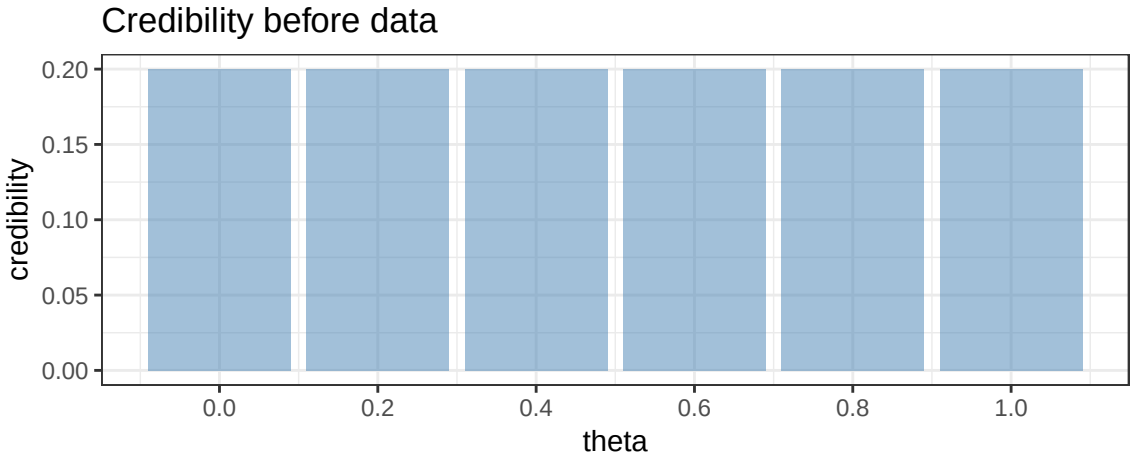
We will make use of a number of R packages as we go along. Here is the code used to load the packages used in this chapter. If you try to mimic the code on your own machine, you will need to use these packages.

```
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 $\theta = 1$.

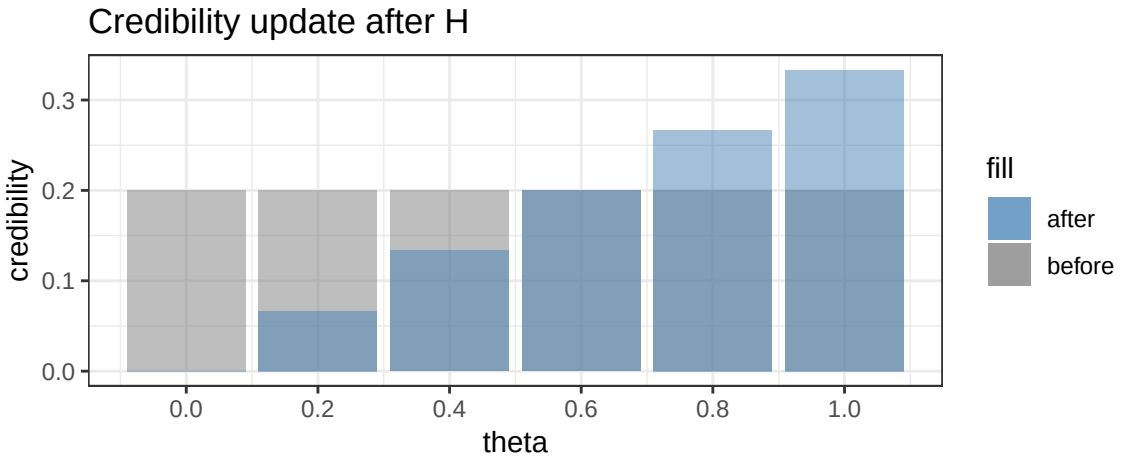
Before collecting our data, if have no other information, we will consider each coin to be equally credible. We could represent that as follows.



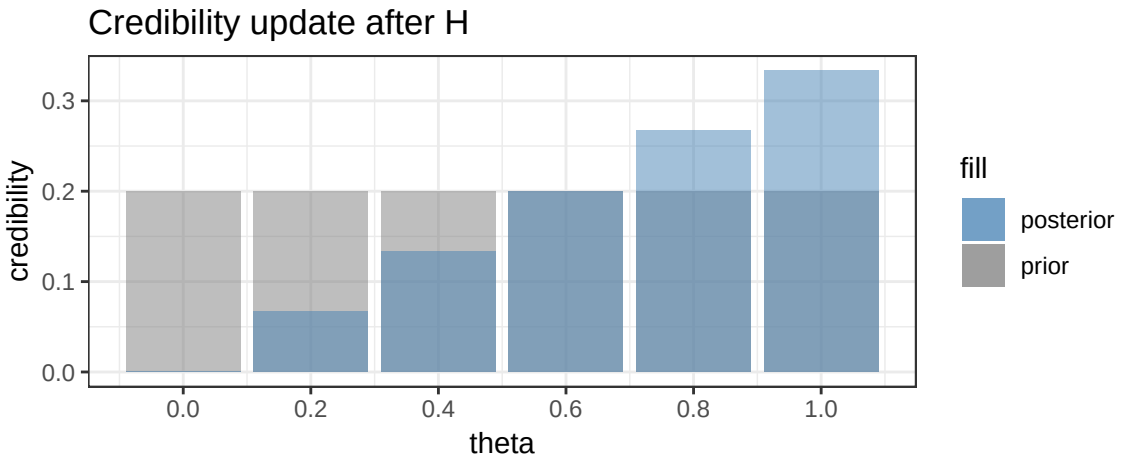
Now suppose we toss the coin and obtain a head. What does that do to our credibilities? Clearly $\theta = 0$ is no longer possible. So the credibility of that option becomes 0. The other credibilities are adjusted as well. We will see later just how, but the following should be intuitive:

- 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%).

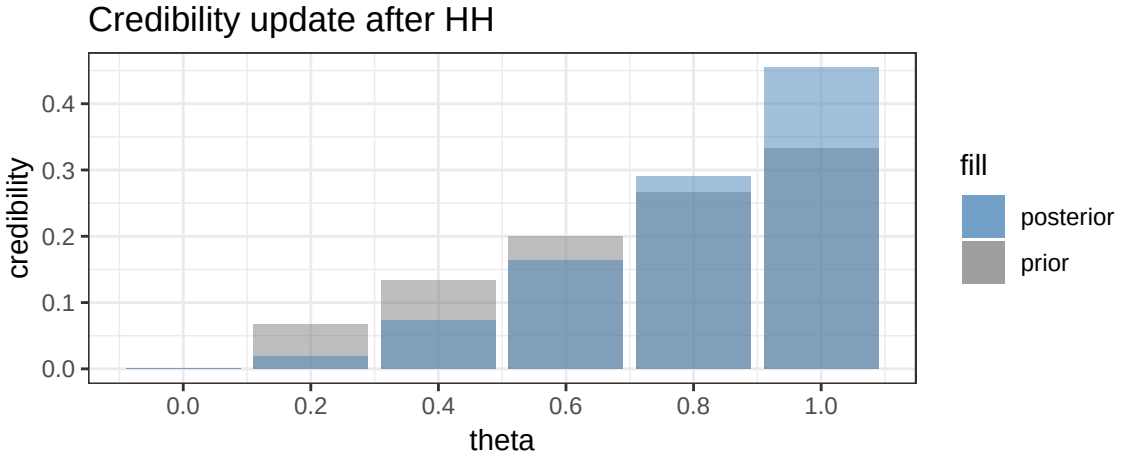
In fact, the adjusted credibility after one head toss looks like this:



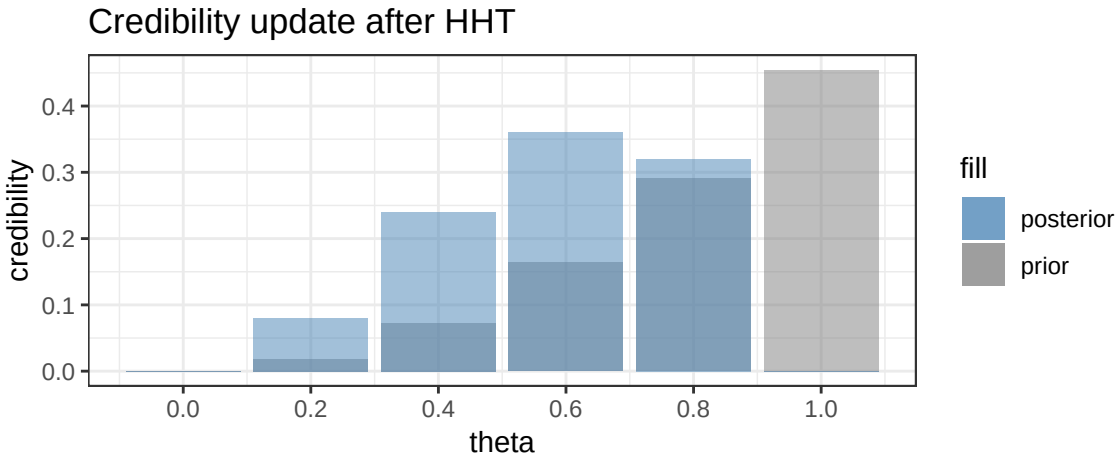
This updating of credibility of possible values of θ is the key idea in Bayesian inference. Bayesians don't call these distributions of credibility "before" and "after", however. Instead they use the longer words "prior" and "posterior", which mean the same thing.



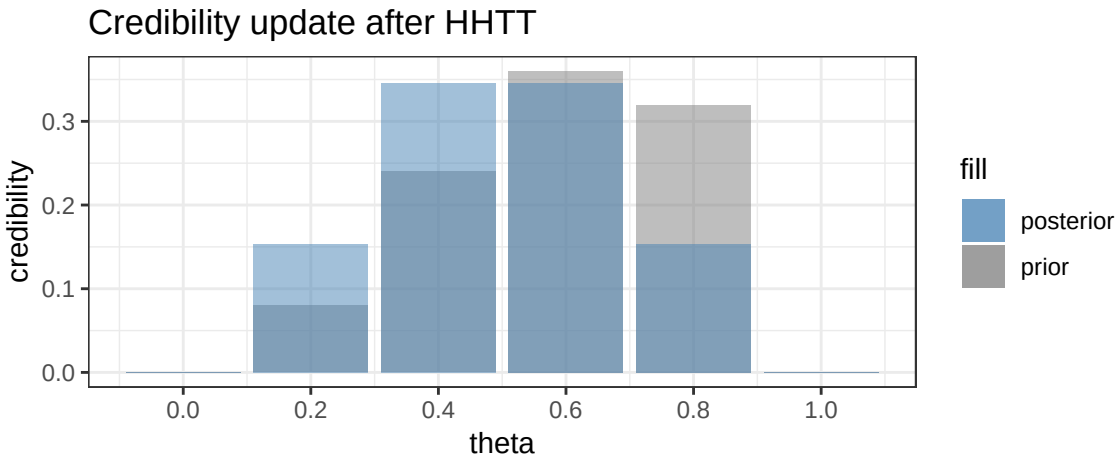
Now suppose we toss the coin again and get another head. Once again we can update the credibility, and once again, the larger values of θ will see their credibility increase while the smaller values of θ will see their credibility decrease.



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.



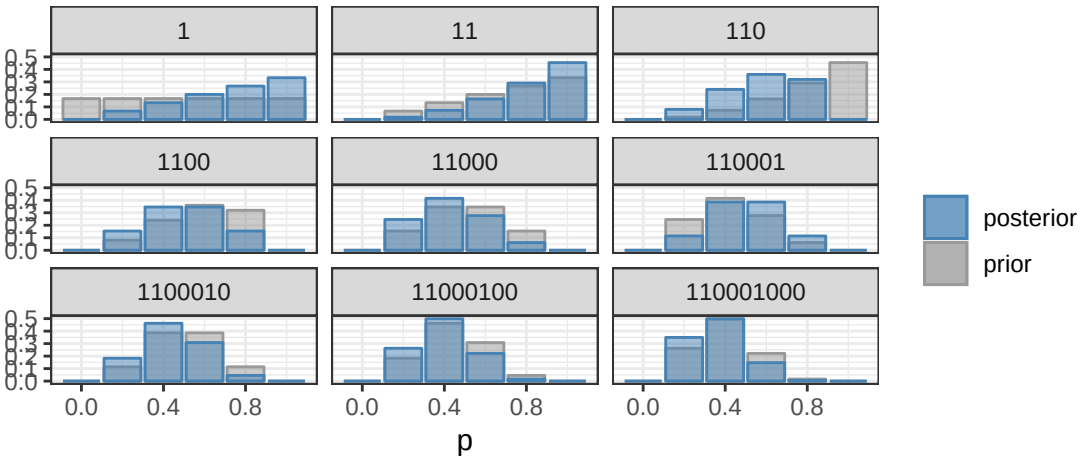
As expected, the posterior is now symmetric with the two central values of θ having the larger credibility.

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
```

[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.

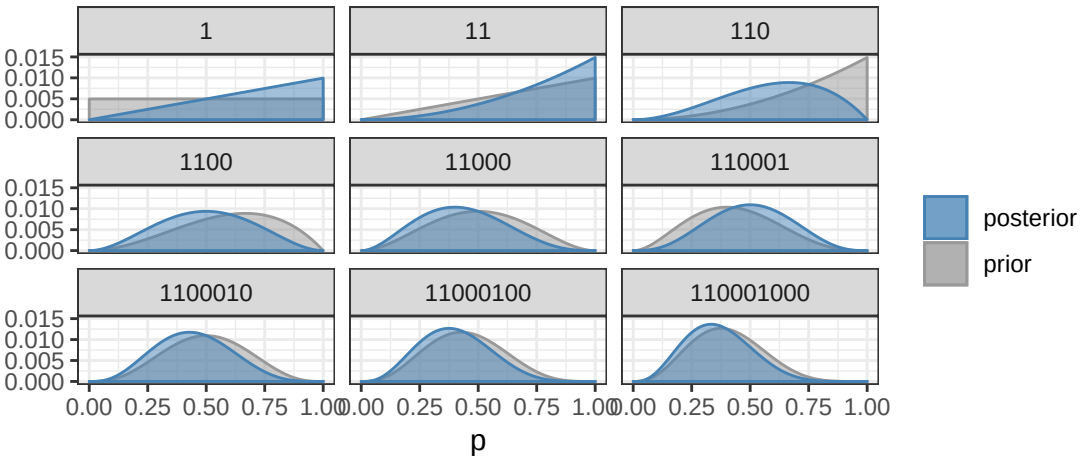


2.2.1 Freedom of choice

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 paramter value.

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

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



2.3 Distribtions

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.

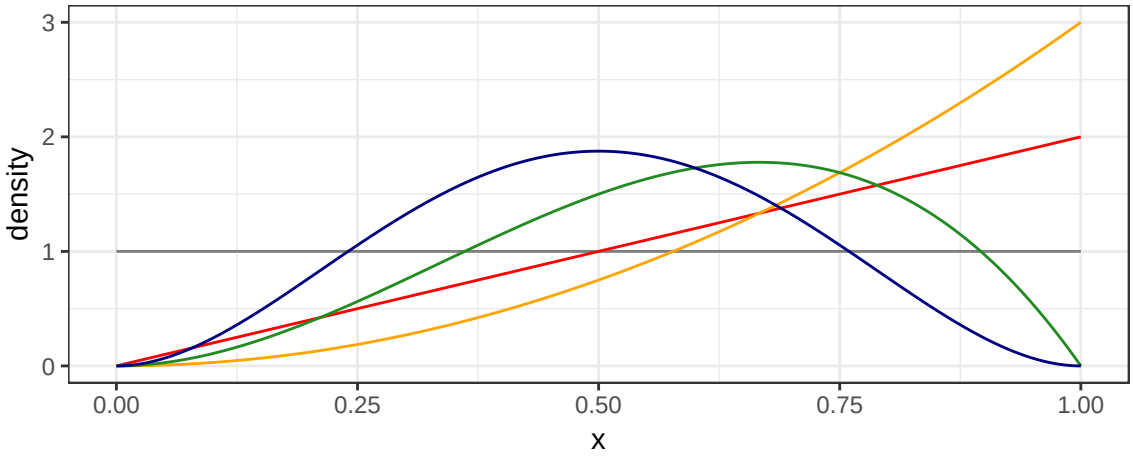
We will also encounter densities with names like "normal", "beta", and "t". The `gf_dist()` function from `ggformula` can be used to plot distributions. We just need to provide R's version of the name for the family

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")
```

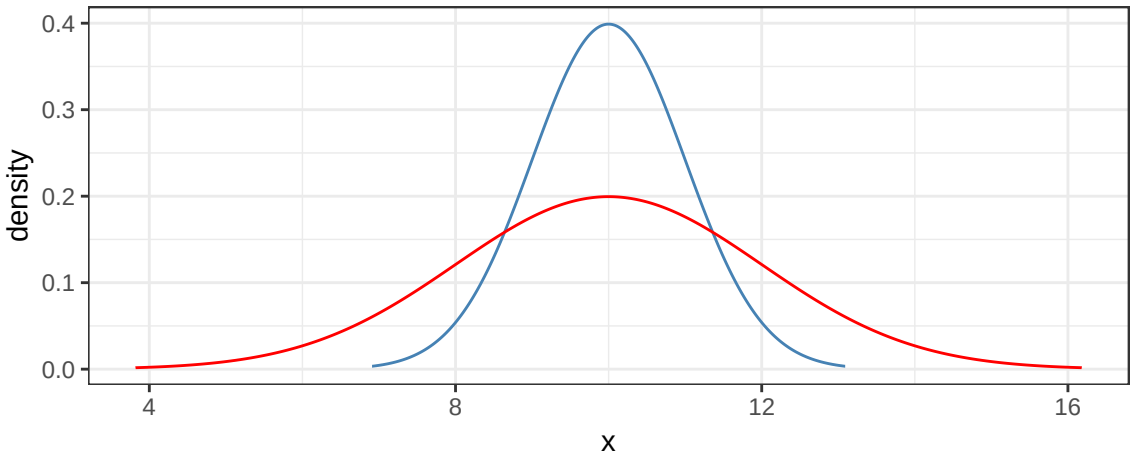


2.3.2 Normal distributions

Another important family of distributions is the normal family. These are bell-shaped, symmetric distributions centered at the mean (μ). A second paramter, 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.

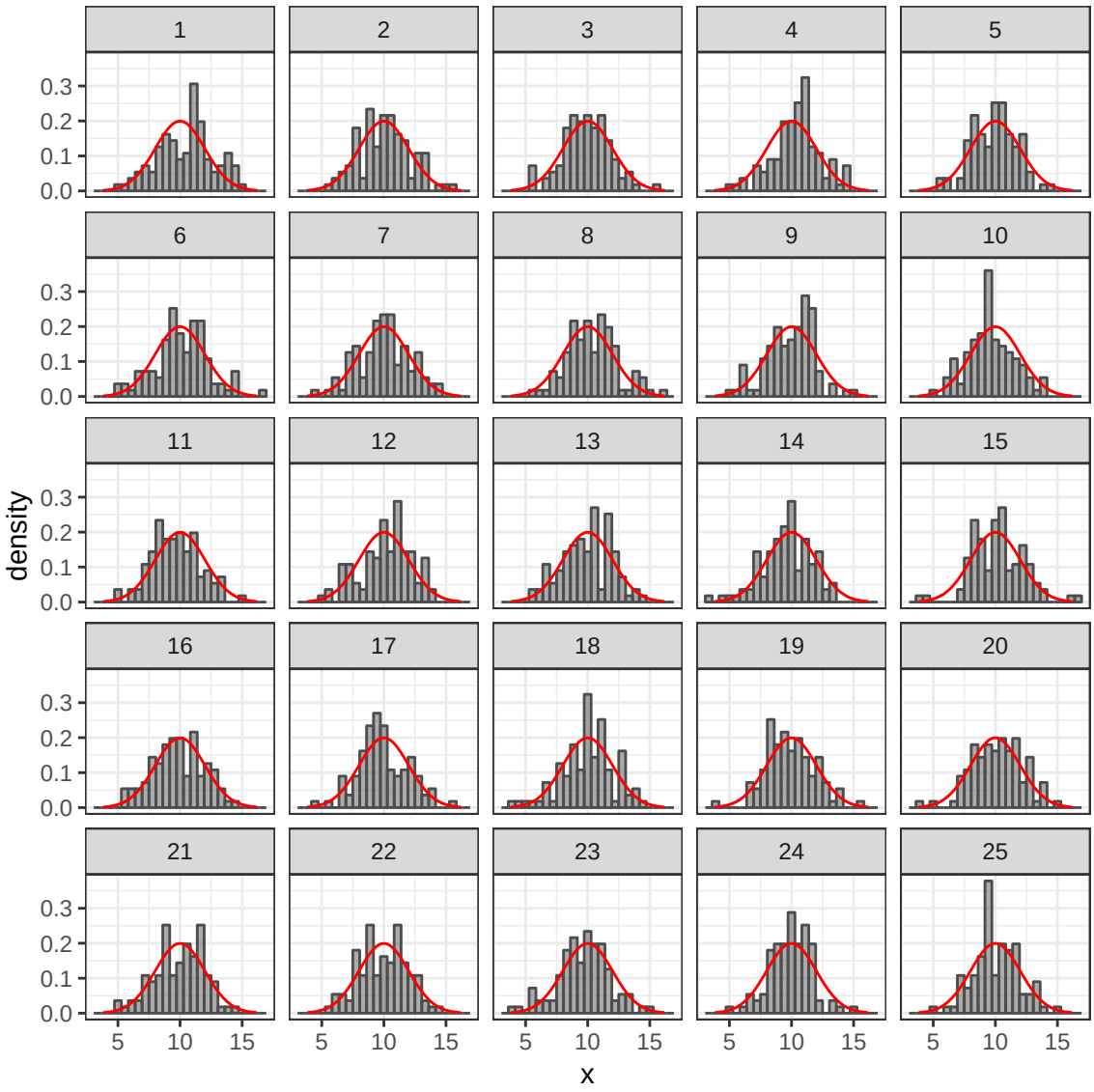
We can also draw random samples from distributions. Random samples will not exactly follow the shape of the distribution they were drawn from, so it takes some experience to get calibrated to know when things are "close enough" to consider a proposed distribution to be believable, and when they are "different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



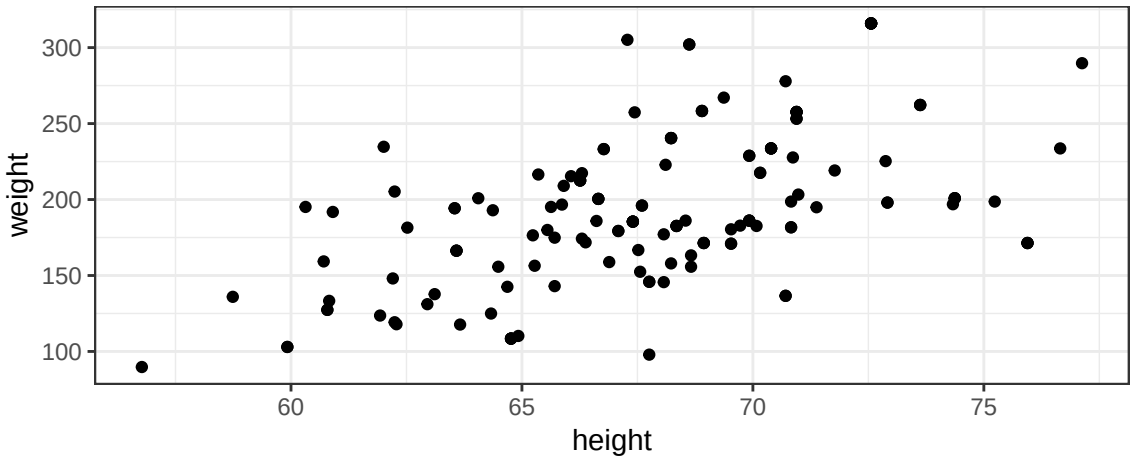
We will see many other uses of these functions. See the next chapter for in introduction to R functions that will be useful.

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 beteween 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 distrubted 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

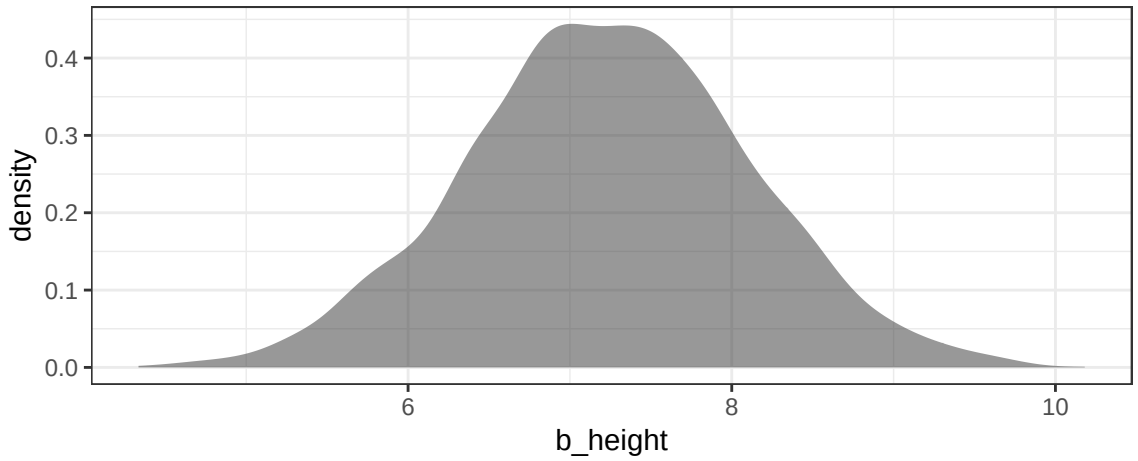
A prior distribution describes what is known/believed about the parameters before we use the information from our data. This could be informed by previous data, or it may be a fairly uniformative prior that considers many values of the parameter to be credible. For this example, we use very flat broad 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 distibutions 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 paramter 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 calcuate 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)
```



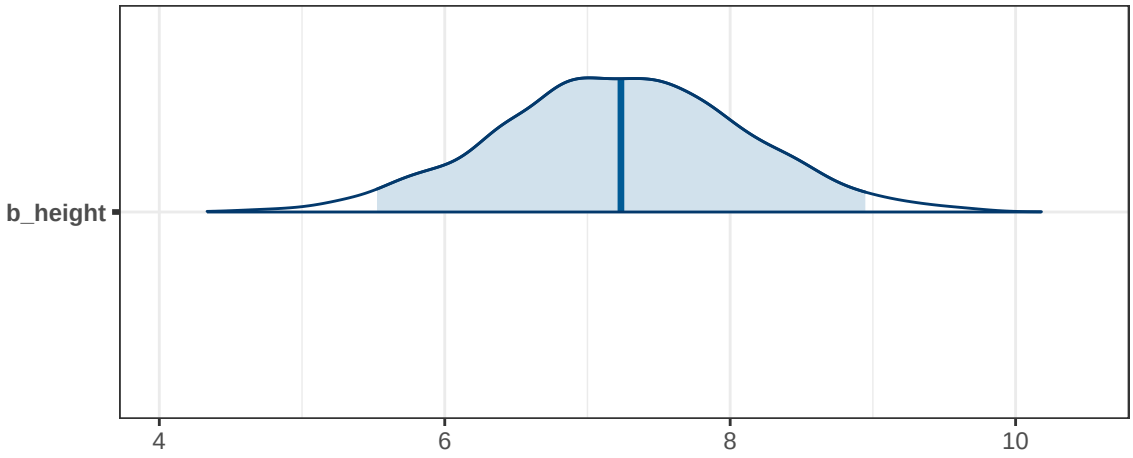

```
mean(~ b_height, data = Post)
```

```
## [1] 7.233
```

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

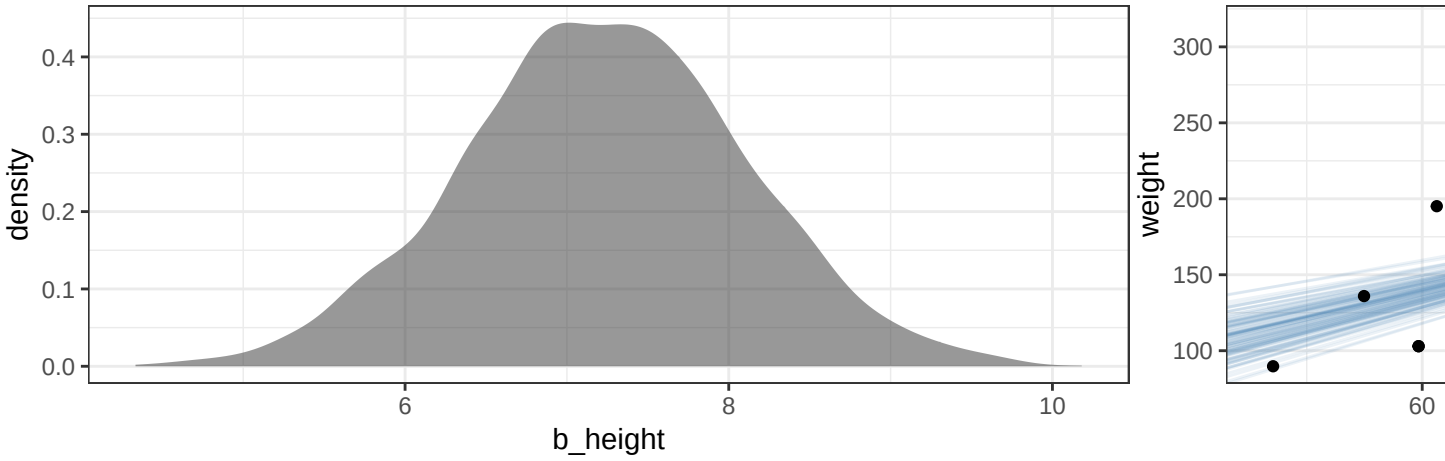
par	lo	hi	prob
b_height	5.535	8.956	0.95

```
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 paramters 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___
-334.8	7.747	39.79	-784.2
-298.7	7.337	42.00	-782.8

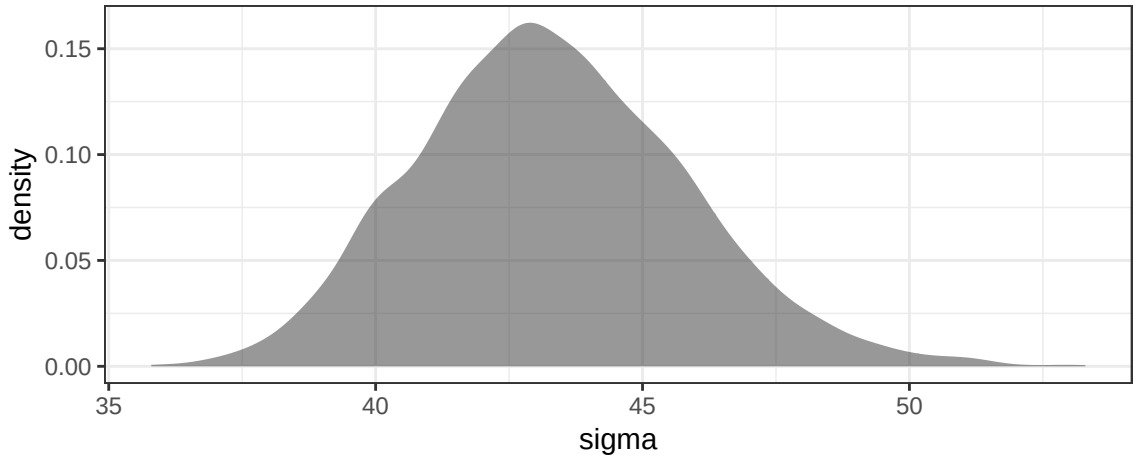
To simulate a weight for a height of 65 inches based on the fist row, we could take a random draw from a $\text{Norm}(-334.8 + 7.747 \cdot 65, 39.79)$ distribution. We can do a similar thing for the second row.

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

b_Intercept	b_height	sigma	lp___	pred_y
-334.8	7.747	39.79	-784.2	260.3
-298.7	7.337	42.00	-782.8	119.2

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)
```

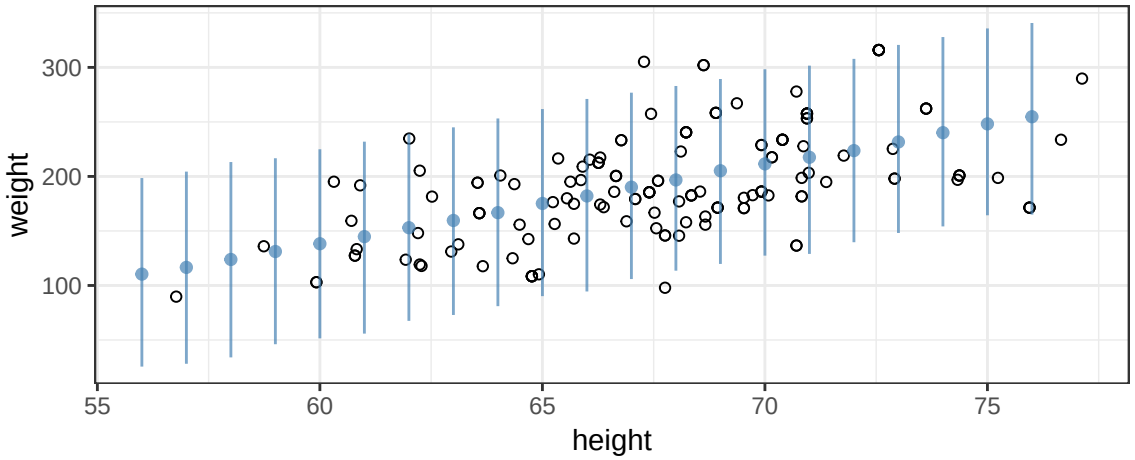


We should not be surprised to see some ($\sim 5\%$) of people who are 85 pounds above or below the average weight for their height.

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 expriment 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) illusrated 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) illusrated 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) illusrated 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) illusrated 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, 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 detailes 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 spacebar.**

   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 spacebar.

   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. **Excpetions** 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 homwork

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 restults 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 habbits 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 dataset 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 varaiables 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 espcially 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 arrangment 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 witout parenthesis, you will see all three parts listed:

```
ilogit
```

```
## function (x)
## {
##   exp(x)/(1 + exp(x))
## }
## <bytecode: 0x7f8200f0ff00>
## <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 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()`]

3.8 Footnotes

Chapter 4

Probability

```
knitr::opts_chunk$set(
  fig.width = 6,
  fig.height = 2.5
)
library(ggformula)
library(dplyr)
library(mosaic)
theme_set(theme_bw())
```

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 outcomess 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.

probabilty mass function (pmf): a fuction (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

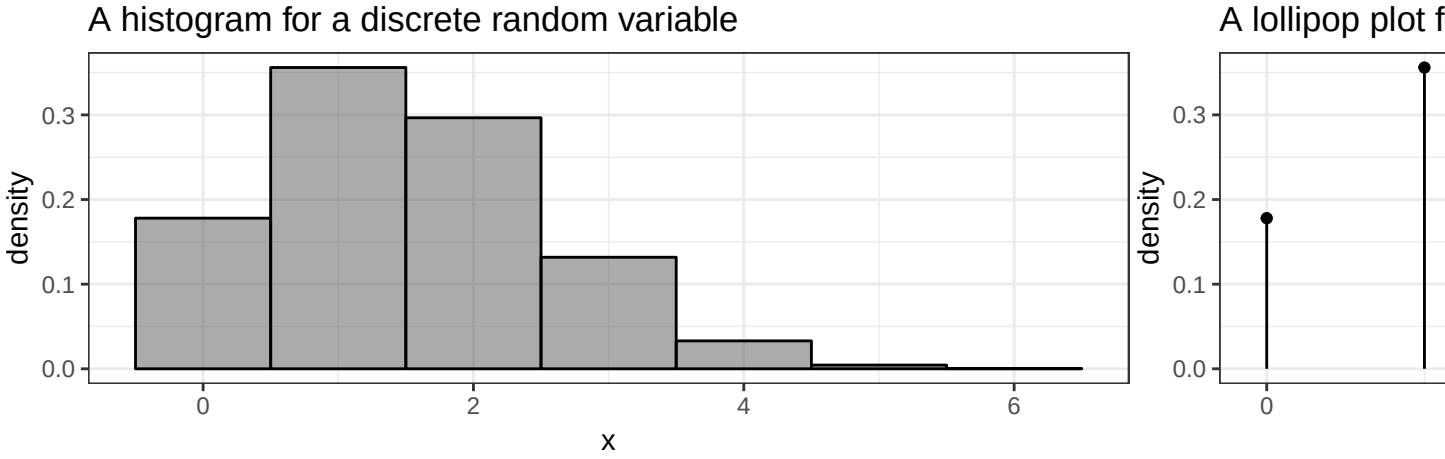
$$\sum_{x \in S} p(x) = 1$$

  and any function with this property is a pmf.

- Probabilites 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 exmaple 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.)

probabilty 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]

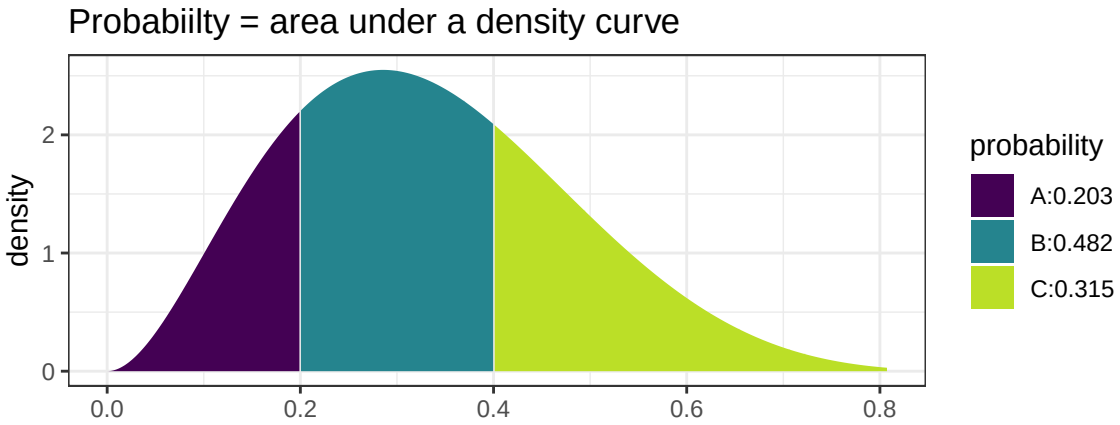
$$\int_{x \in S} f(x)\, dx = 1$$

  and any function with this property is a pmf.

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

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

[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.



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 probabilty 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 **discrte 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.

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` follwed 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 exmple, 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.

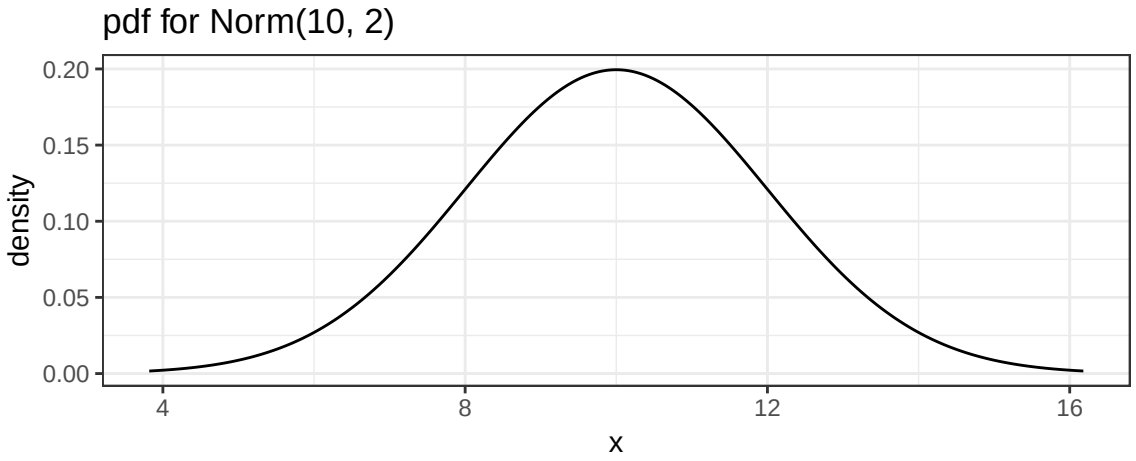
```
args(dnorm)
```

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

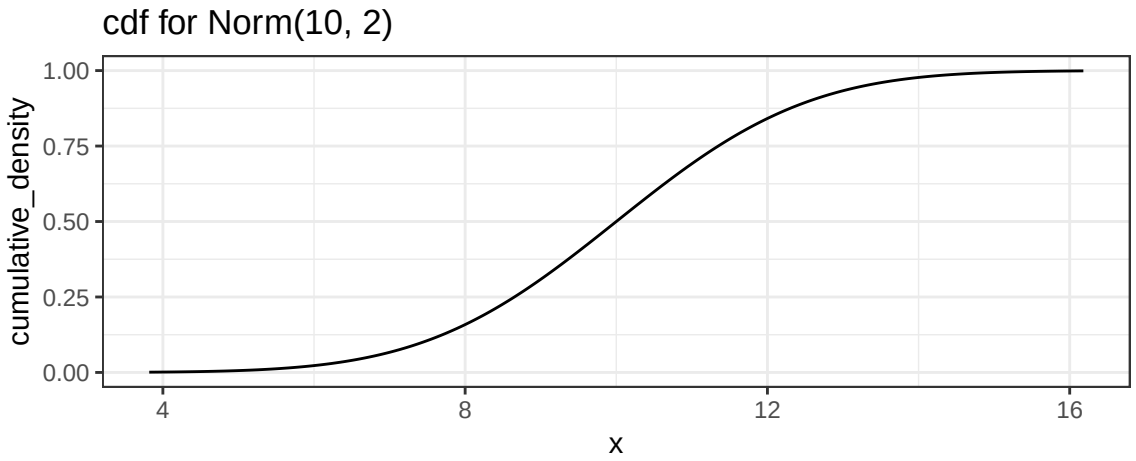
From this we see that the parameters are called `mean` and `sd` and have default value of 0 and 1. These values will be used if we don't specify somethng else. As with many of the pmf and pdf functions, there is also an option to get back the log of the pmf or pdf by setting `log = TRUE`. This turns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

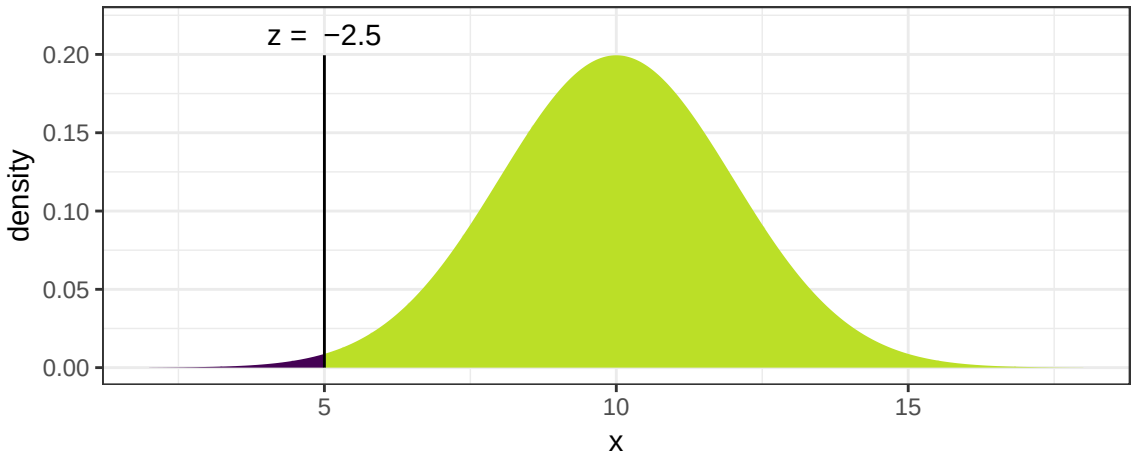
```
pnorm(5, mean = 10, sd = 2)
```

```
## [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?

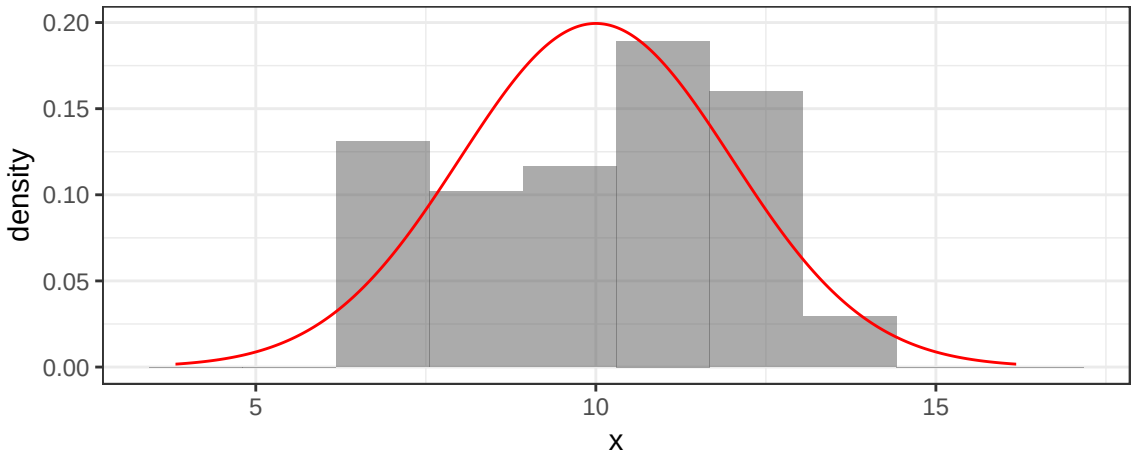
   Again, we know the answer is the square of the standard deviation, so 4. But let's get R to compute it in a way that would work for other distributions as well.

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

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

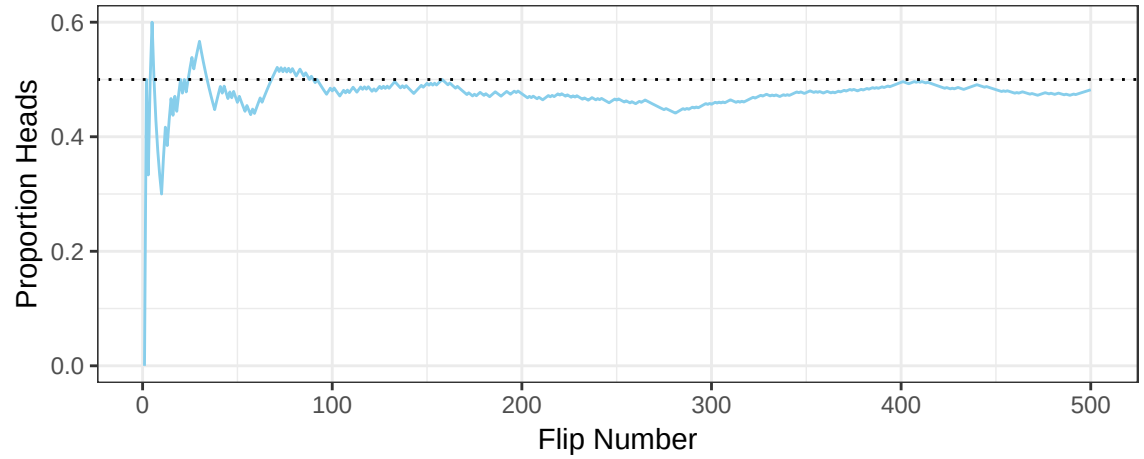
6. Simulate a data set with 50 values drawn from a $\mathsf{Norm}(10, 2)$ distribution and make a histogram of the results and overlay the normal pdf for comparison.

```
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 me 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. $p(\text{blue} \mid \text{blond})$.
   c. $p(\text{blond} \mid \text{blue})$.

[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 also stored in a different format (as a 3d table rather than as a data frame).

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 joing 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 indendence 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 equivlent 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

4.5 Footnotes

Chapter 5

Bayes' Rule and the Grid Method

5.1 The Big Baysian 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

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

Furthermore, the computation of the posterior seems straightforward since, letting D be the data values and θ the parameter values, we have

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

Notice that the denomiator sums or integrates over all possible numerators, so it is just a normalizing constant that guarntees that total probability is 1 for the posterior distribution.

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:

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

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
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

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

5.2.1 Creating a Grid

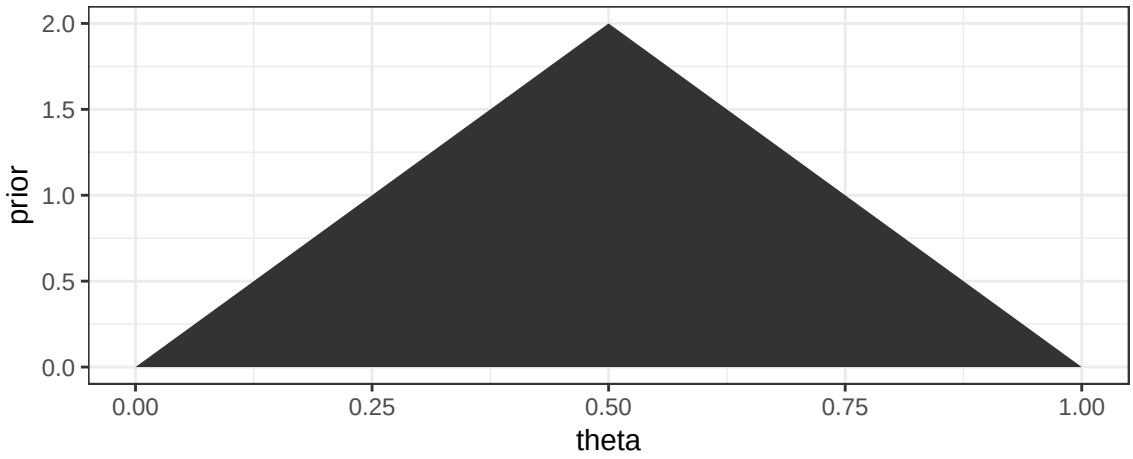
The paramter is θ and we will discretize by selection 1001 grid points from 0 to 1 by 0.001.

```
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 the prior

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

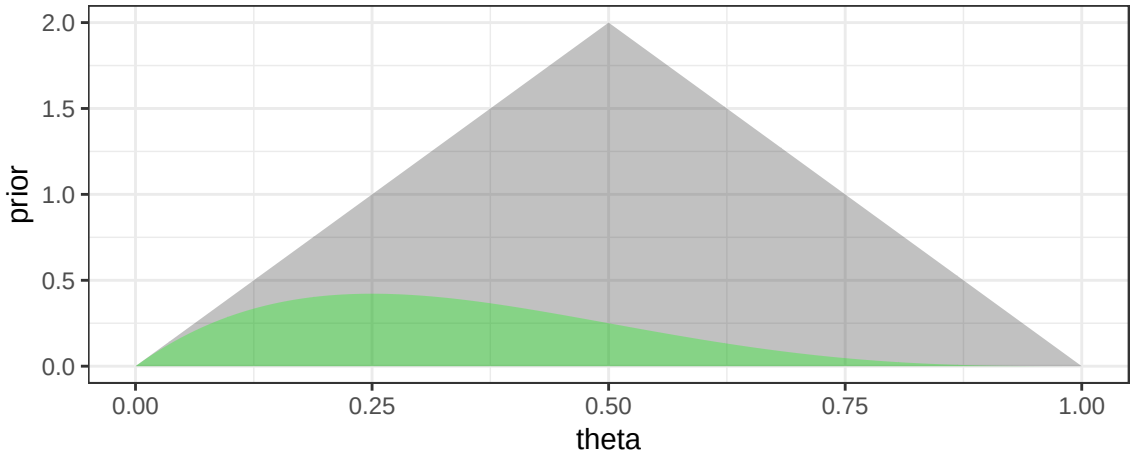


Now the likelihood for a small data set: 1 success out of 4 trials. This is the trickiest part. `dbinom(x, size = n, prob = theta)` will calculate the probability that we want for a given value of `x`, `n`, and `theta`. We want to do this

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

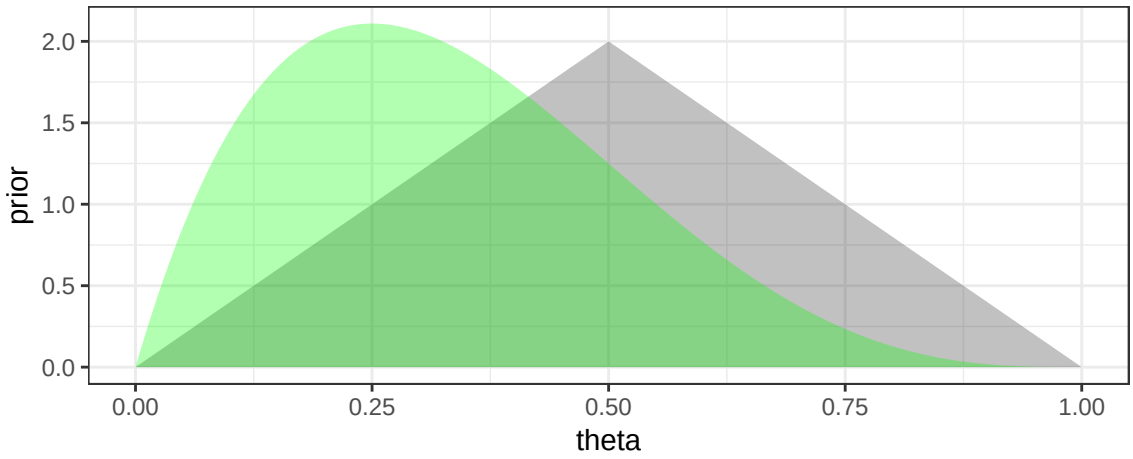
`purrr:map_dbl()` helps us tell R how to do this. Each value of `theta` gets plugged in for `.x` and a vector of numbers (dbl stands for double – computer talk for real number) is returned.

```
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")
```



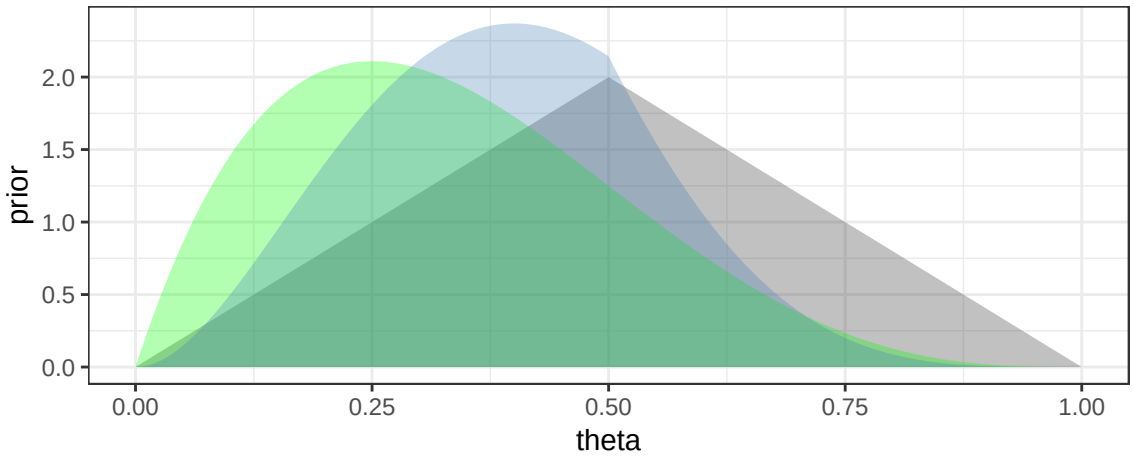
Note: Likelihoods are NOT pmfs or pdfs, so the total area under a likelihood function is usually not 1. We can make a normalized version for the purpose of plotting. (Recall, we will normalize the posterior at the end anyway, so it is fine if the likelihood is off by a constant multiple at this point 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")
```

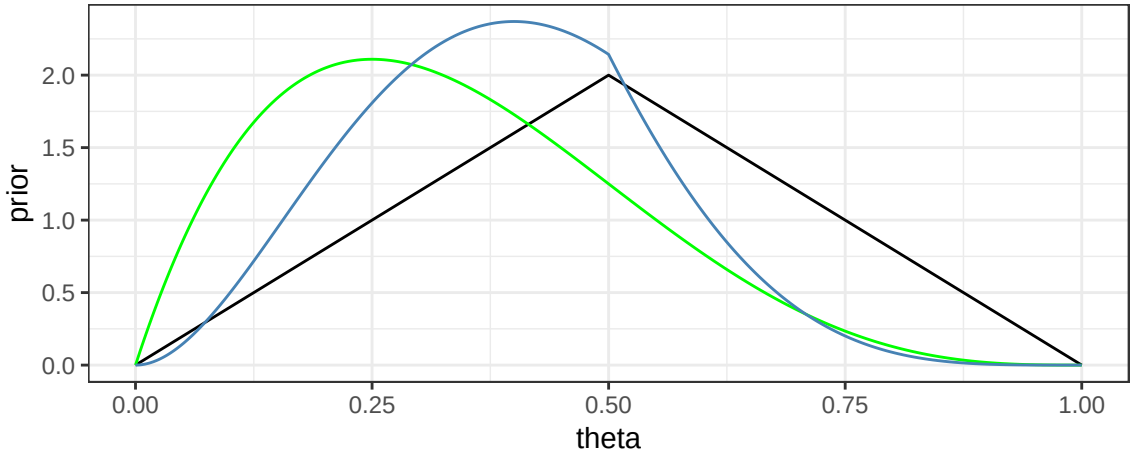


The hardest part of the coding (computing the likelihood) is now done. Getting the posterior is as simple as computing a product.

```
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

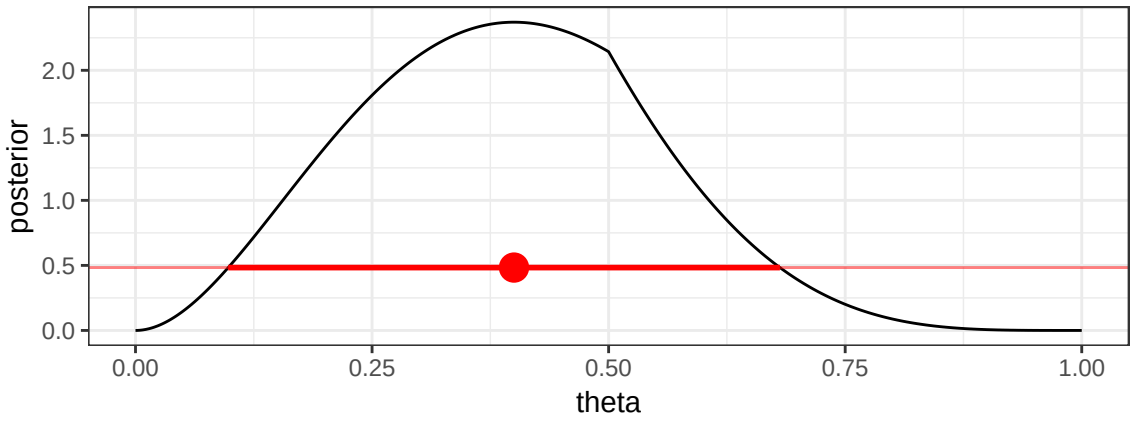
- the data frame containing our grid calculations,
- `pars`: the name of the parameter (or parameters) for which we want intervals (default is the first column in the grid),
- `prob`: the probability we want in covered by our interval (0.95 by default),
- `posterior`: the name of the column containing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



posterior mode and 95% HPI indicated in red

5.2.3 Automating the grid

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

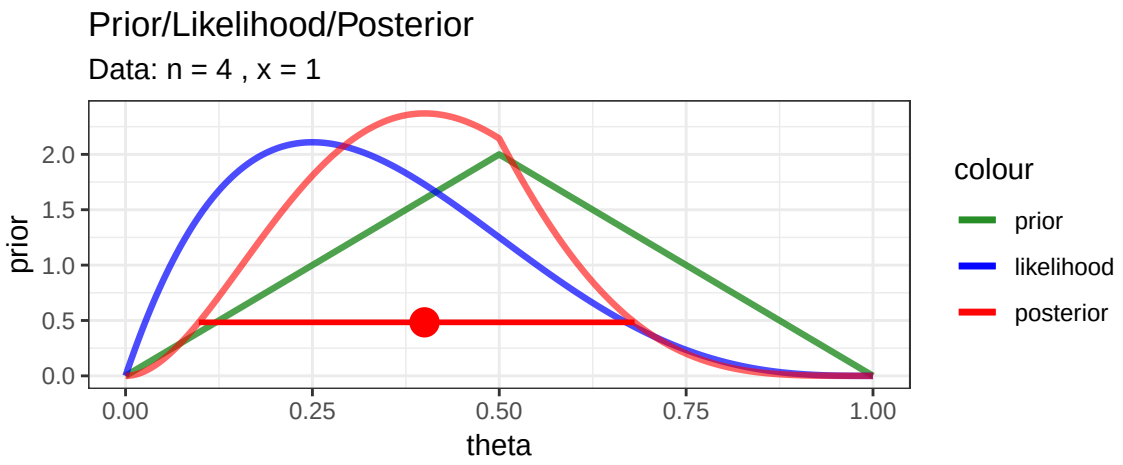
```
MyBernGrid <- function(
  x, n,                    # x successes in n tries
  prior = dunif,
  resolution = 1000,       # number of intervals to use for grid
  ...) {

  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
}
```

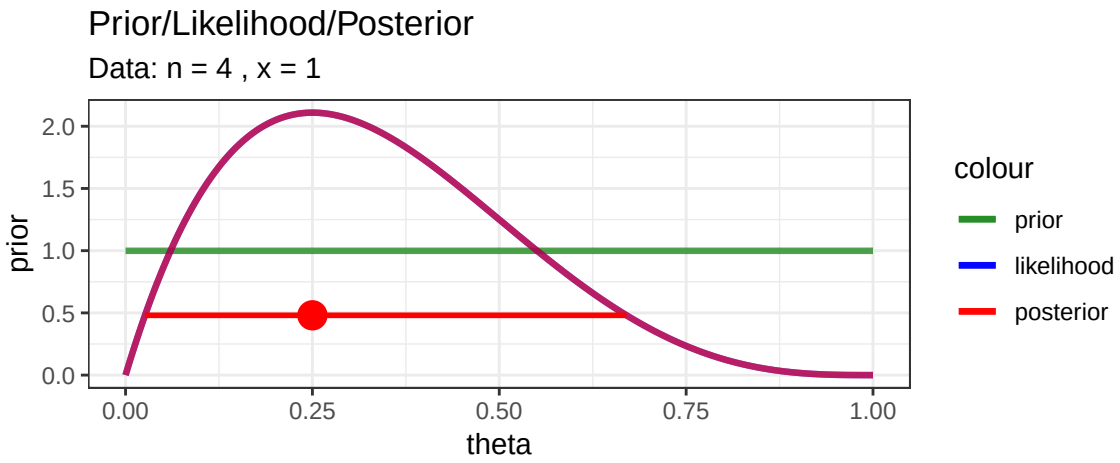
This function let's us quickly explore several scenarios and compare the results.

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

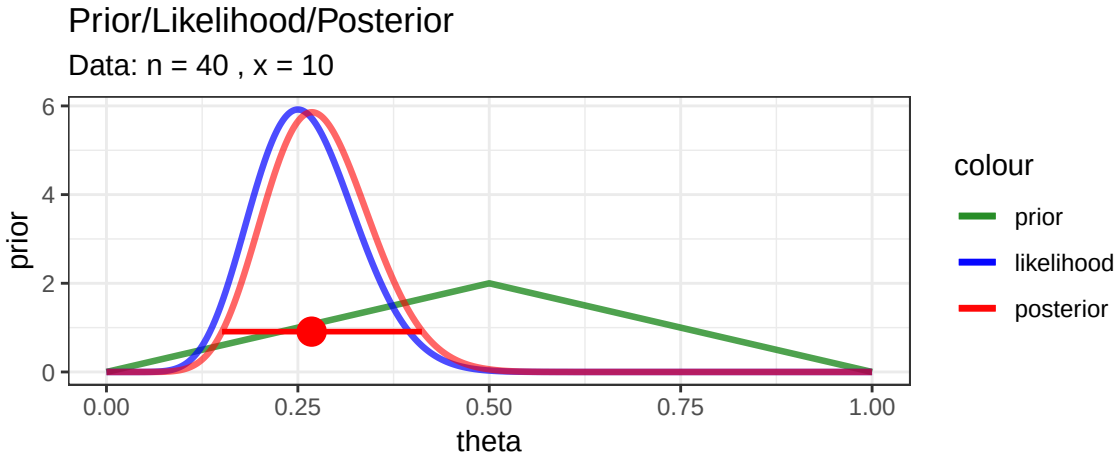
```
library(triangle)
MyBernGrid(1, 4, prior = dtriangle, a = 0, b = 1, c = 0.5)
```



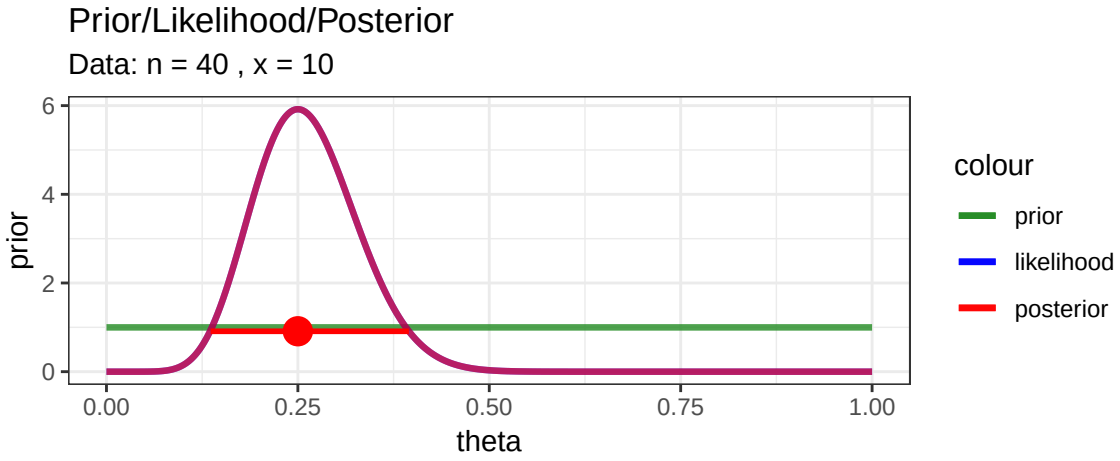

```
MyBernGrid(1, 4, prior = dunif)
```



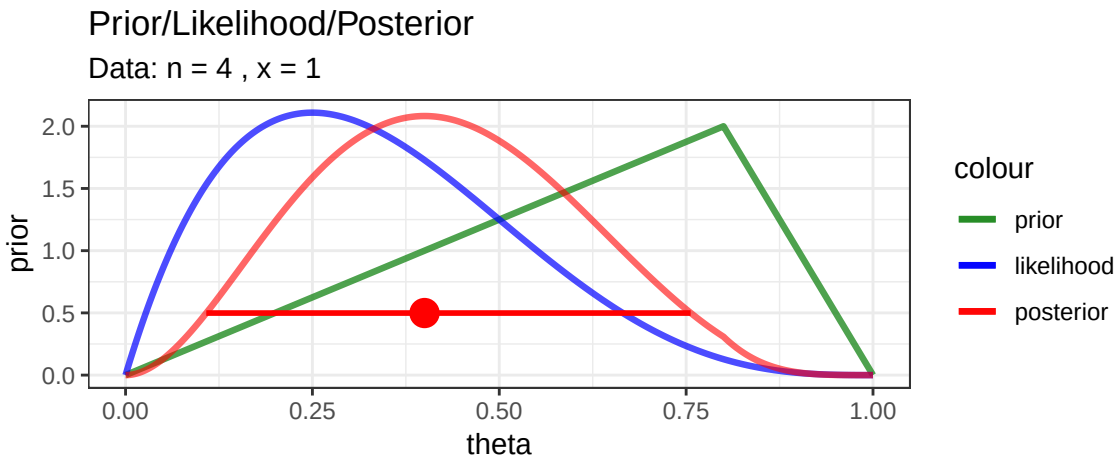

```
MyBernGrid(10, 40, prior = dtriangle, a = 0, b = 1, c = 0.5)
```



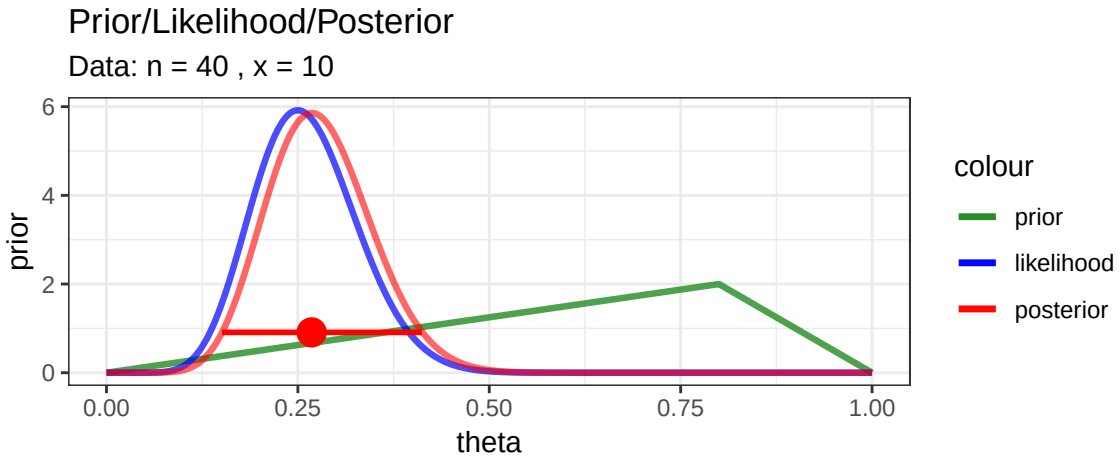

```
MyBernGrid(10, 40, prior = dunif)
```



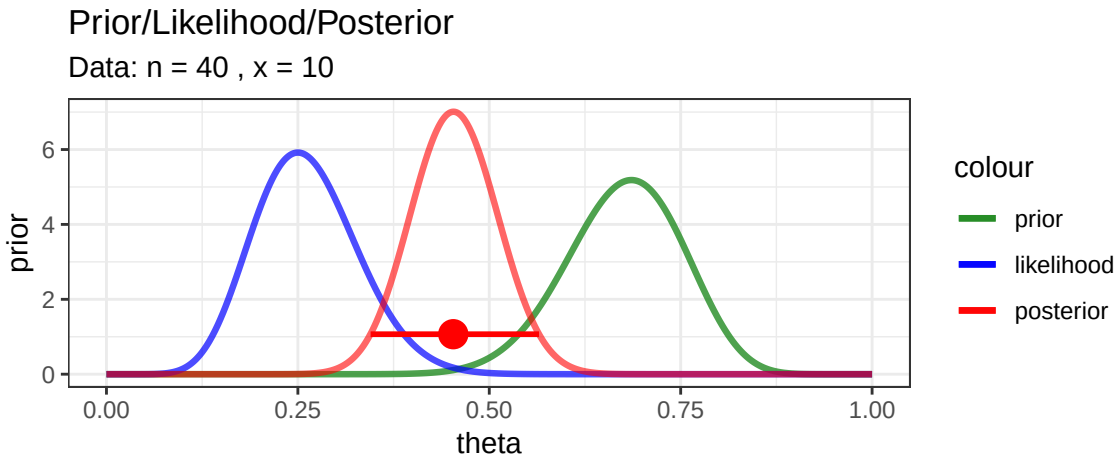

```
MyBernGrid(1, 4, prior = dtriangle, a = 0, b = 1, c = 0.8)
```




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




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



5.3 Exercises

1. Suppose we have a test with a 99% hit rate (sensitivity) and a 5% false alarm rate (95% specificity) just like in the example on page 103 of *DBDA2e*. 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, 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.

2. 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?

3. Consider again the disease and diagnostic test of the previous two exercises.

   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. How does the result compare with your answer to Exercise 5.1?

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

Part II

Inferring a Binomial Probability

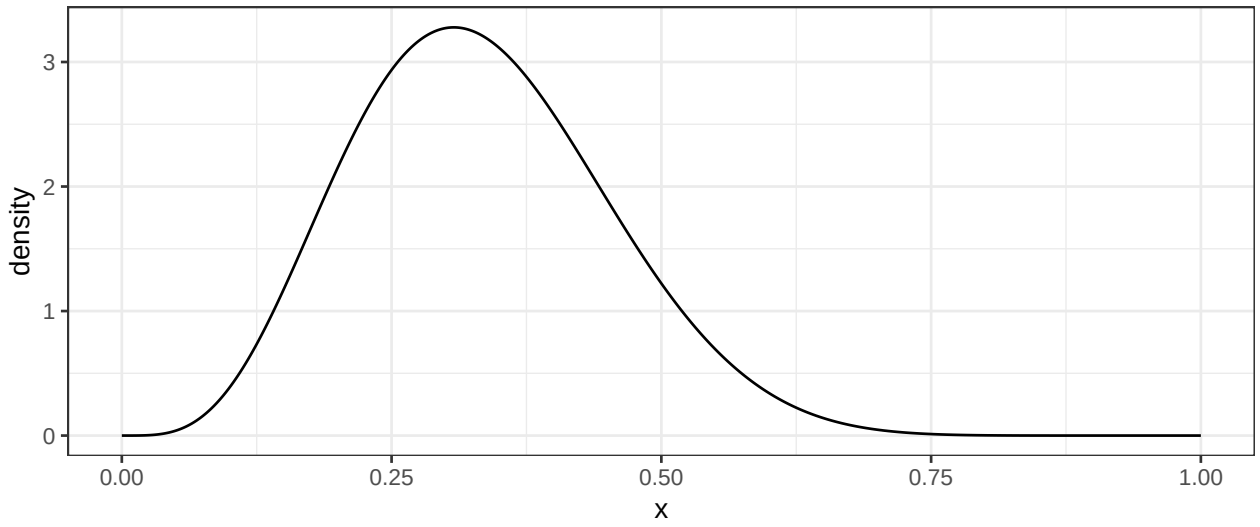
Chapter 6

Inferring a Binomial Probability via Exact Mathematical Analysis

6.1 Beta distributions

6.1.1 You oughta be in pictures

```
gf_dist("beta", shape1 = 5, shape2 = 10)
```



6.1.2 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	Beta(,)
Parameters	> 0 shape (real)
	> 0 shape (real)
Support	$x \in [0, 1]$ or $x \in (0, 1)$

PDF	$\frac{x^{\alpha-1}(1-x)^{\beta-1}}{\mathrm{B}(\alpha,\beta)}$
Mean	$\frac{\alpha}{\alpha+\beta}$
Mode	$\frac{\alpha-1}{\alpha+\beta-2}$ for $\alpha,\beta>1$ 0 for $\alpha=1,\beta>1$ 1 for $\alpha>1,\beta=1$
Variance	$\frac{\alpha\beta}{(\alpha+\beta)^2(\alpha+\beta+1)}$
Concentration	$\kappa=\alpha+\beta$

6.1.3 Alternative Parametizations

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

6.1.3.1 Mode and concentration

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

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

6.1.3.2 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

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

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

6.1.4 beta_params()

`CalvinBayes::beta_params()` will compute several summaries of a beta distribution given any of hese 2-parameter summaries. This can be very handy for coverting 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

6.1.5 Automating Bayesian updates for a proportion (beta prior)

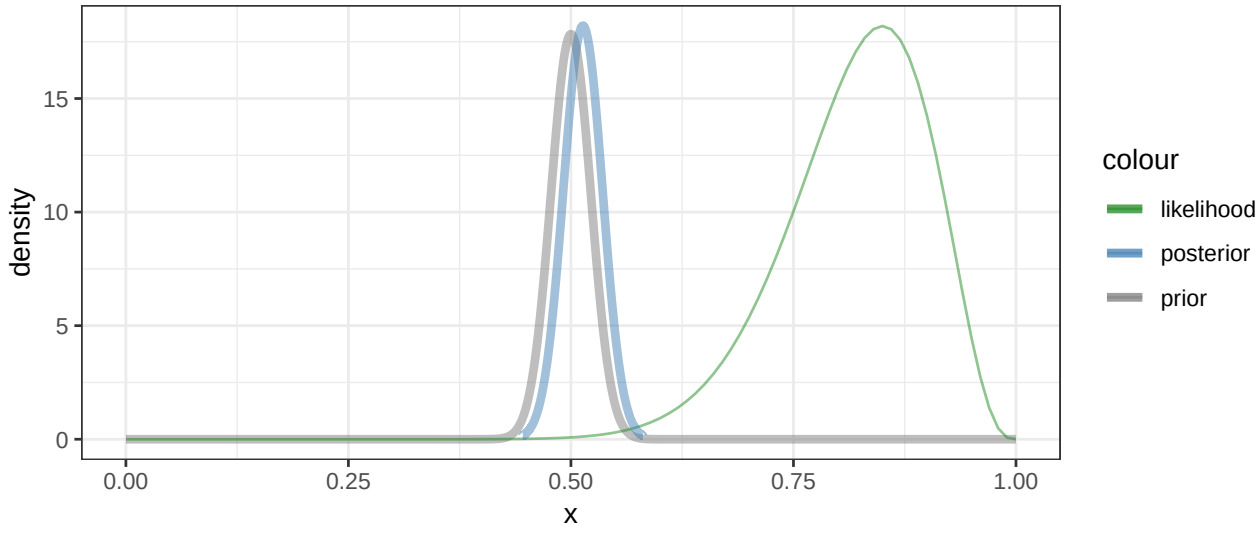
Since we have formulas for this case, we can write a function handle any beta prior and any data set very simply. (Much simpler than doing the grid method each time).

```
quick_bern_beta <-
  function(
    x, n,      # data, successes and trials
    a, b,      # shape parameters for beta distribution
    ...        # see clever trick below
  )
  {
    if (missing(a) || missing(b)) {
      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
    }
```

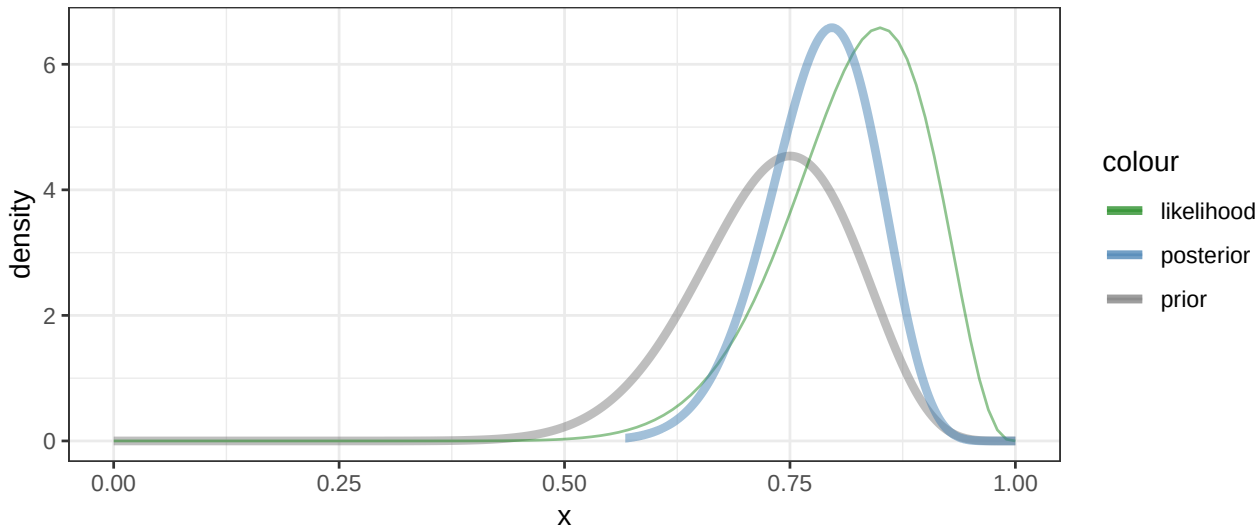
```
    gf_dist("beta", shape1 = a, shape2 = b,
            color = ~ "prior", alpha = 0.5, xlim = c(0,1), size = 1.6) %>%
      gf_function(likelihood,
                  color = ~ "likelihood", alpha = 0.5) %>%
      gf_dist("beta", shape1 = a + x, shape2 = b + n - x,
              color = ~ "posterior",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

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

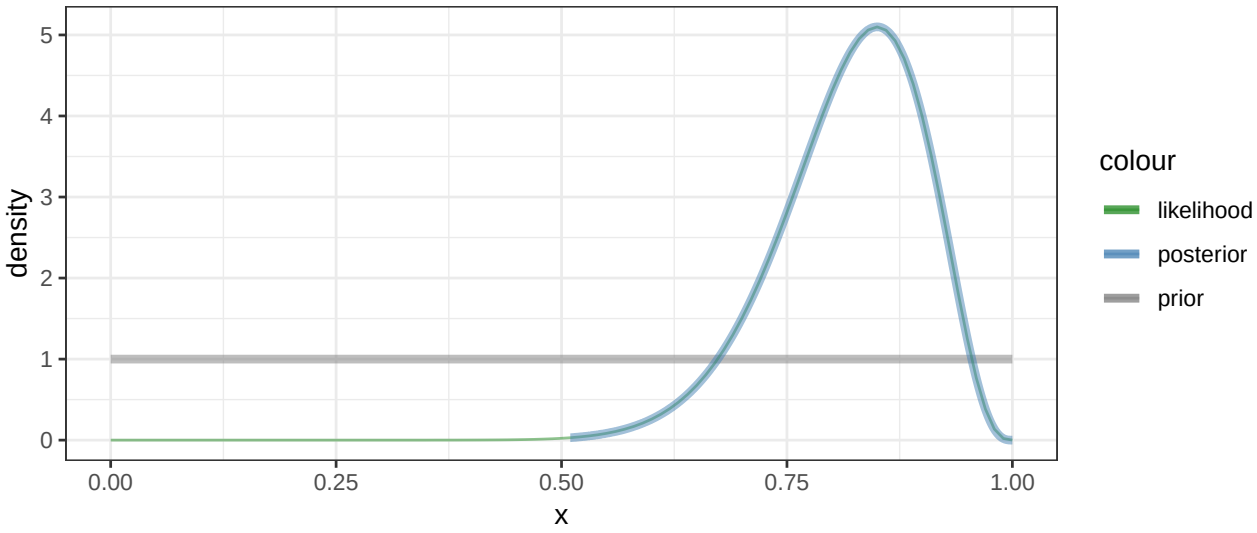
```
quick_bern_beta(17, 20, mode = 0.5, k = 500)
```




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




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

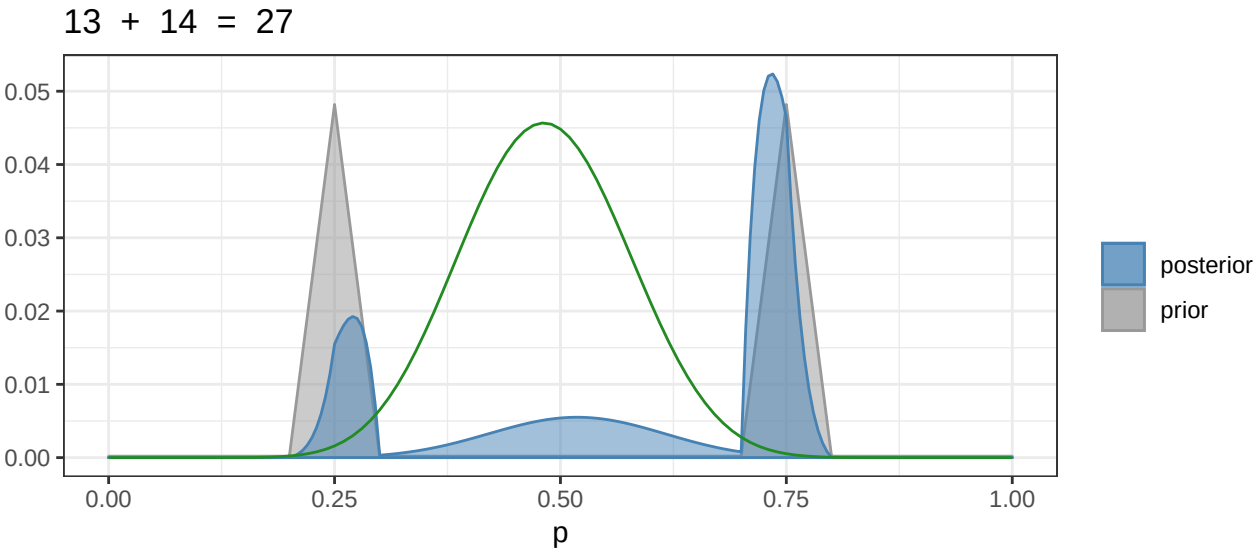


6.2 What if the prior isn't a beta distribution?

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")
```



6.3 Exercises

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

   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 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 new 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 Advisor 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.

Example: 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 poplations 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 Advisor 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
- doesn't require him to remember the populations of all the islands

He can ask the clerk on any island what their 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, randomly pick one of the 4 other islands (a proposal island) and travel there in the morning, inquiring about the population over lunch.
  - 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).
- Before leaving, enquire about the population of the current island.
- When arriving at the proposal island, equire 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 probabilty R, else return to the "current" island (ie, last night's island).

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

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

- What happens if $R = 1$?
- What happens if $R = 0$?

Somehow R must depend on the populations of the current and proposal islands. But how? If R is too large, the king will visit small islands too often. If R 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 R.

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 $P(X_t = x)$?

If

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

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**:

$$P(X_{t+1} = x_j \mid X_t = x_i) = p_{ij}$$

That is, the probability of moving from state i to state j is one step is the same at every step.

7.2.4 Matrix representation

A time-homogeneous Markov Chain can be represented my 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

- 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?

The Metropolis Algorithm:

- 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 transition from island a to island b? ($P(X_{t+1} = b \mid X_t = a)$)

7.2.5 Regular Markov Chains

A time-homegeneous 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

Small Example:

- Is our small example regular? How many steps are required?

Metropolis Algorithm:

- 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 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 R so that

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

7.3 Back to King Markov

If R is betwen 0 and 1, and the jumping rule allows us to get to all the islands (eventaully), 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 satified at time $t+1$ ($wM = w$):

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

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

$$P(a \to_t b) = P(b \to_t a)$$

Why does this work?

- Suppose $P(X_t = a) = f(a)$ as the law prescribes.
- $P(a \to_t b) = P(b \to_t a)$ makes the joint distribution symmetric: For any a and any b.

$$P(X_t = a, X_{t+1} = b) = P(X_t = b, X_{t+1} = a)$$

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

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

- 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 R? Recall the ingredients:

- $F(a)$ be the population of island a
- $f(a)$ be the proportion of the total popluation living on island a ($f(a) = \frac{F(a)}{\sum_x F(x)}$)
- $J(b \mid a)$ is the conditional probability of selecting island b as the candidate when a is the current island. (J for Jump probability)

Consider two islands – a and b – with $F(b) > F(a)$. Let's calculate the probabilities of moving between the islands.

- Probability of moving from a to $b = f(a)J(b \mid a) \cdot 1$
- Probability of moving from b to $a = f(b)J(a \mid b) \cdot R$.

How do we choose J and R to make these probabilities equal?

Answer: We need to choose a jump rule ($J()$) so that the Markov chain is regular (ie, so that is it is possible to get from any island to any island in some fixed number of steps). Given such a $J()$, we can let R be defined by

$$R = \frac{f(a)J(b \mid a)}{f(b)J(a \mid b)}$$

An especially easy case is when $J(b \mid a) = J(a \mid b)$. In that case

$$R = \frac{f(a)}{f(b)}$$

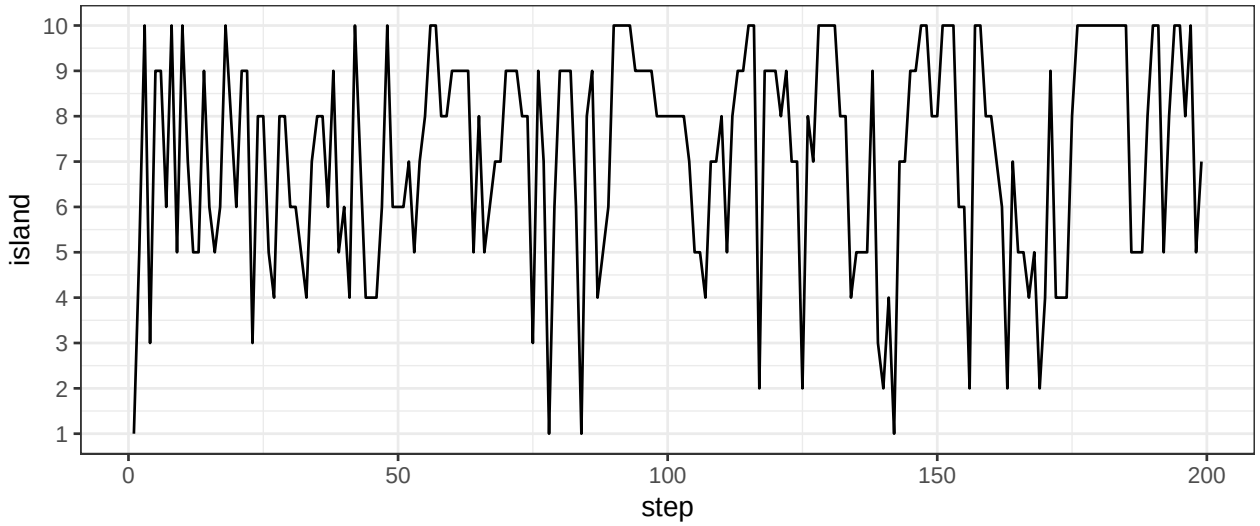
The original Metropolis algorithm used symmetric jump rules. The later generalization (Metropolis-Hastings) employed non-symmetric jump rules to get better performance of the Markov Chain.

7.4 How well does the Metropolis Algorithm work?

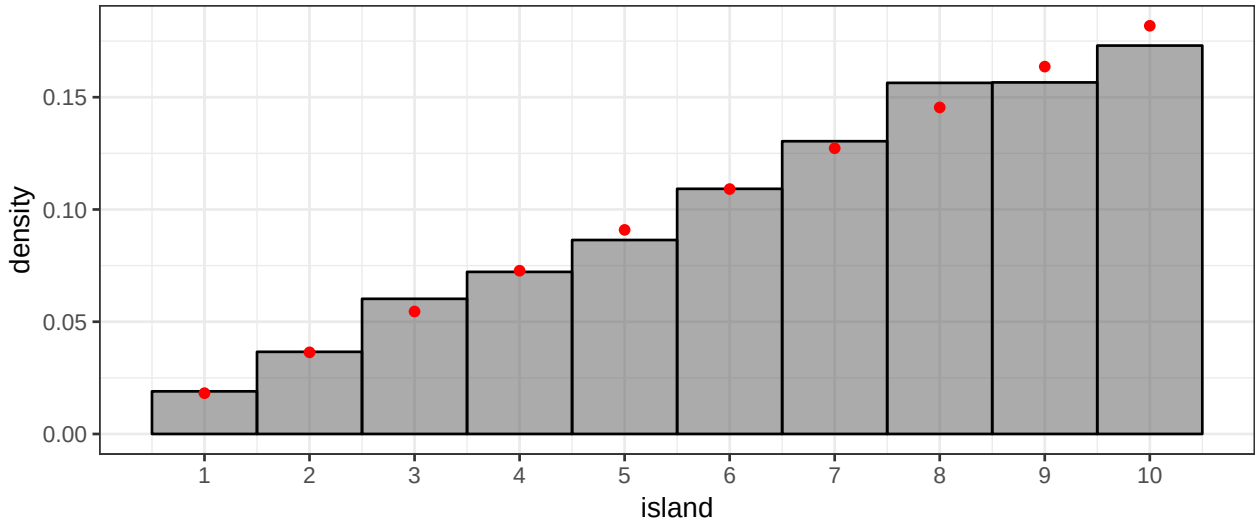
```
KingMarkov <- function(
  num_steps = 1e5,
  population = 1:10,
  start = 1,
  J = function(a,b) 1/(length(population) - 1)
  ) {

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

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

    # propose any 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]
    current <- ifelse(runif(1) < prob_move, proposal, current)
  }
  tibble(
    step = 1:num_steps,
    island = island
  )
}
```

7.4.1 Jumping to any island

```
Tour <- KingMarkov(5000)
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))
```

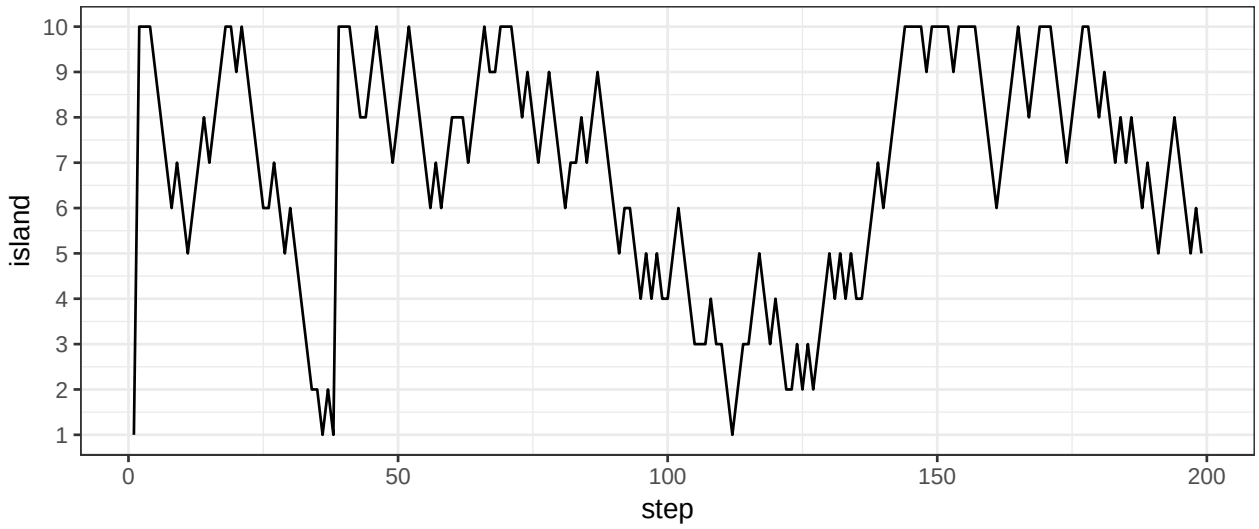



```
gf_dhistogram( ~ island, data = Tour, binwidth = 1, color = "black") %>%
  gf_point(prop ~ island, data = Target, color = "red") %>%
  gf_refine(scale_x_continuous(breaks = 1:10))
```

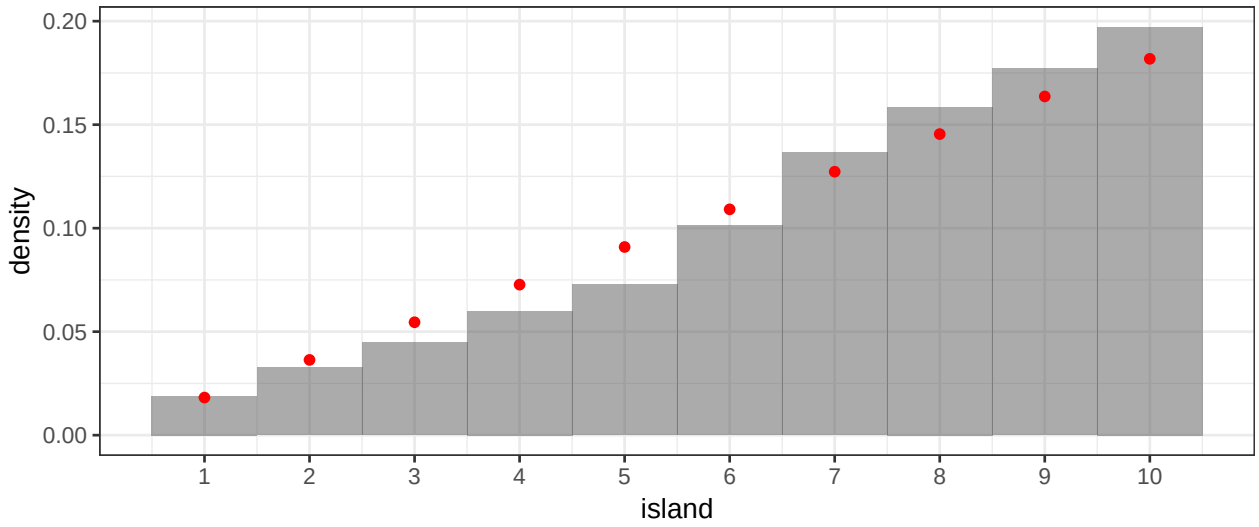


7.4.2 Jumping only to neighbor islands

```
neighbor <- function(a, b) as.numeric(abs(a-b) %in% c(1,9))
Tour <- KingMarkov(5000, 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 princple, 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

$$R = \frac{\text{proposal "posterior"}}{\text{current "posterior"}}$$

Metropolis-Hastings variation:

- More choices for $J()$ gives more opportunity to tune for convergence

Other variations:

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

7.5.1 Estimating mean and variance

Consider the following simple model:

- $y \sim Norm(\mu, \sigma)$
- $\mu \sim Norm(0, 1)$
- $\log(\sigma) \sim Norm(0, 1)$

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

```
MetropolisNorm <- function(
  num_steps = 1e5,
  y,
  size = 1,  # sd of jump distribution
  start = list(mu = 0, log_sigma = 0)
  ) {

  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)
    log_sigma[i + 1] <- rnorm(1, log_sigma[i], size)
    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 = size
  )
}
```

7.5.2 Assessing how well the Algorithm worked, and how to tune

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

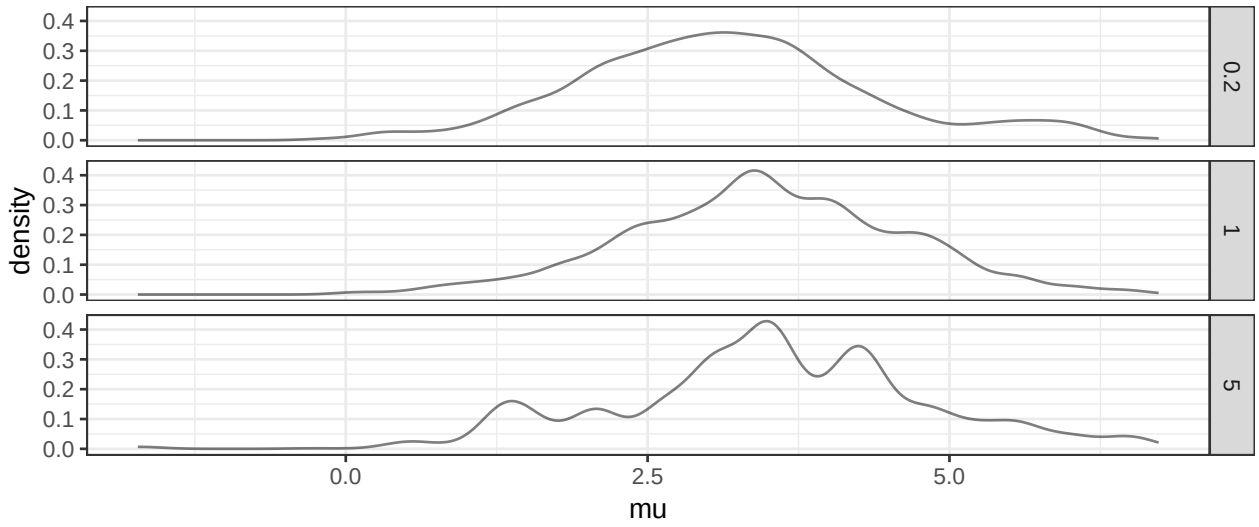
```
y <- rnorm(25, 10, 3)
Tour1 <- MetropolisNorm(y = y, num_steps = 5000, size = 1)
Tour0.2 <- MetropolisNorm(y = y, num_steps = 5000, size = 0.2)
Tour5 <- MetropolisNorm(y = y, num_steps = 5000, size = 5)
Tours <-  bind_rows(Tour1, Tour0.2, Tour5)
```

```
prop(~ move | size, data = Tours)
```

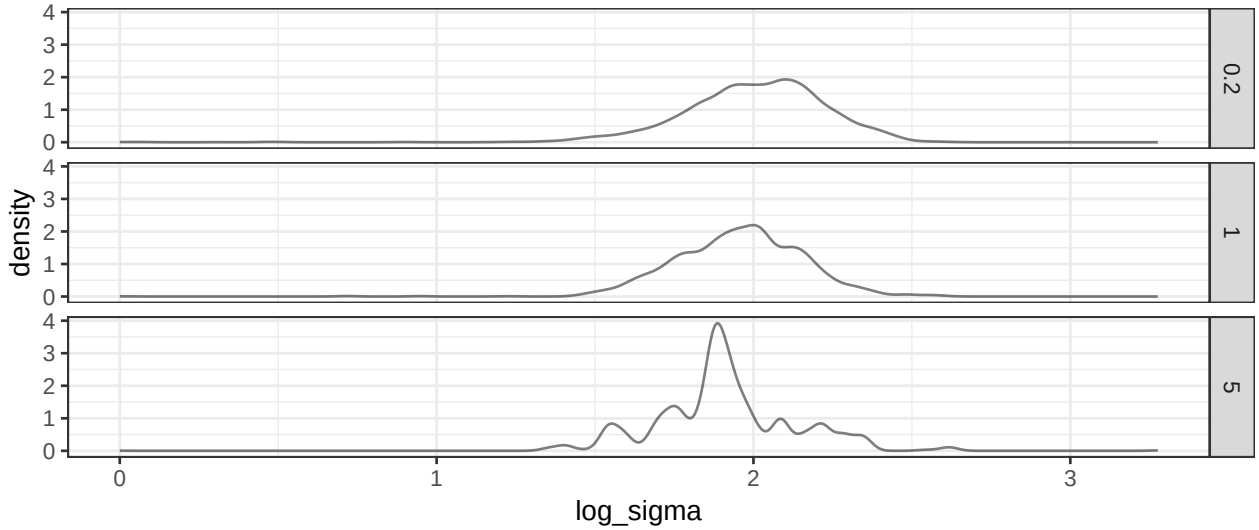
```
## prop_TRUE.0.2   prop_TRUE.1   prop_TRUE.5
##        0.5900        0.1388        0.0122
```

7.5.2.1 Density plots

```
gf_dens( ~ mu | size ~ ., data = Tours)
```

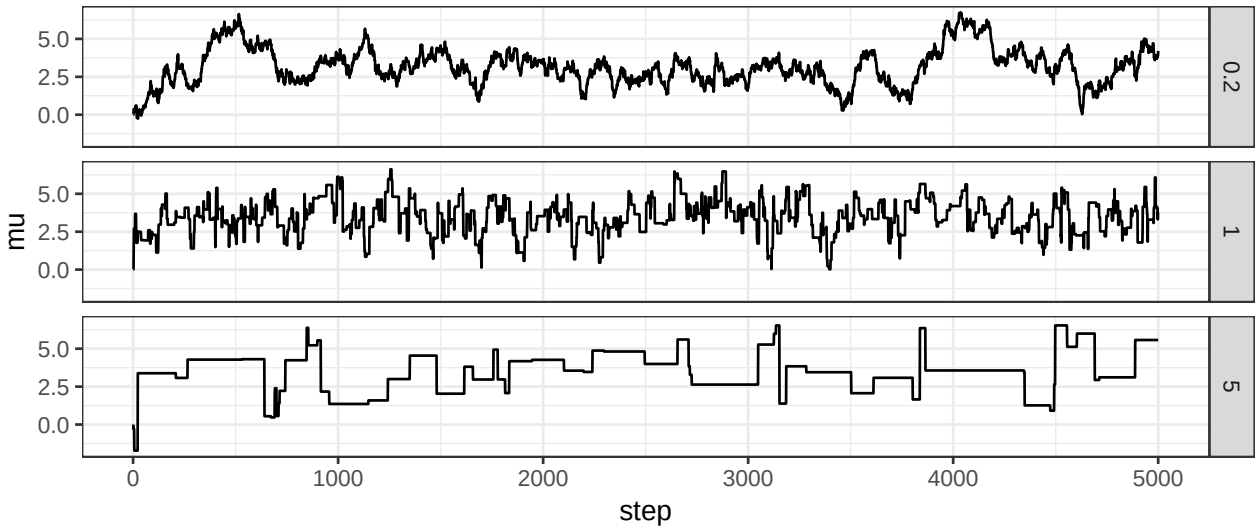



```
gf_dens( ~ log_sigma | size ~ ., data = Tours)
```

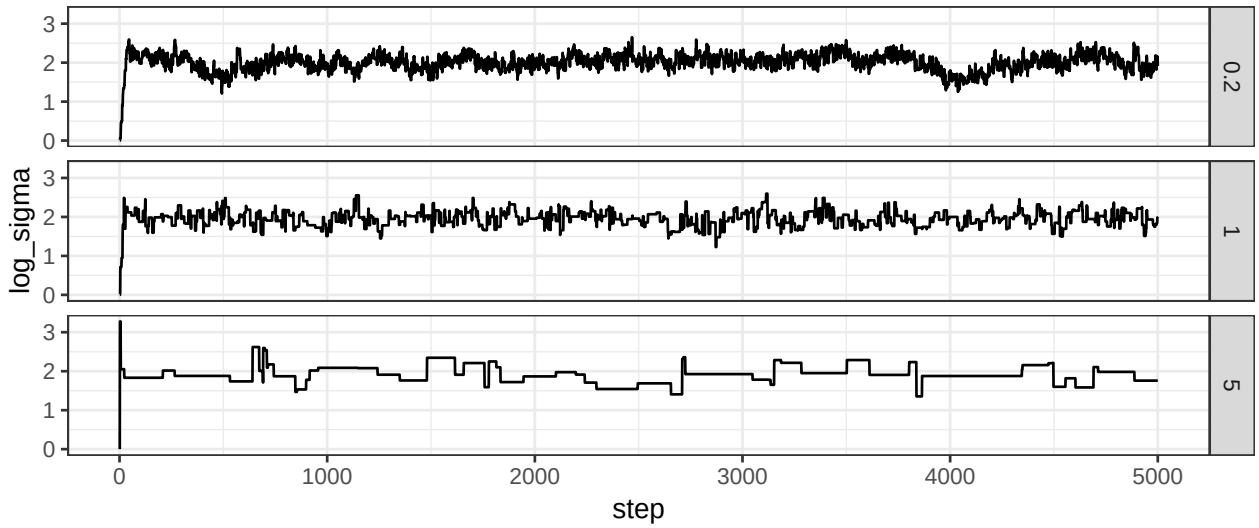


7.5.2.2 Trace plots

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




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

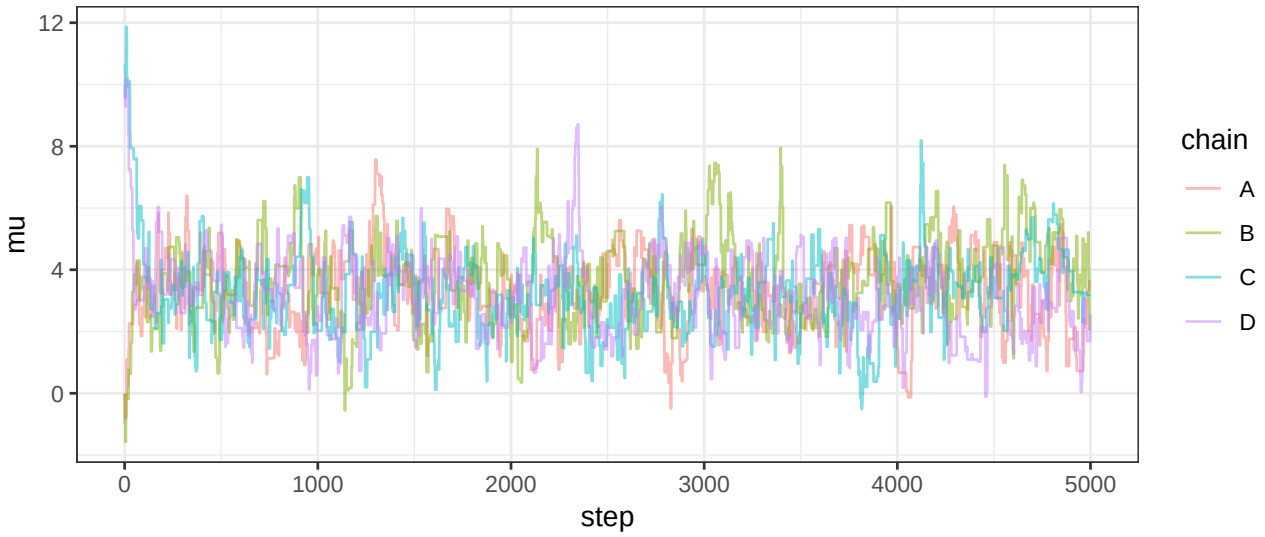


7.5.2.3 Comparing Multiple Chains

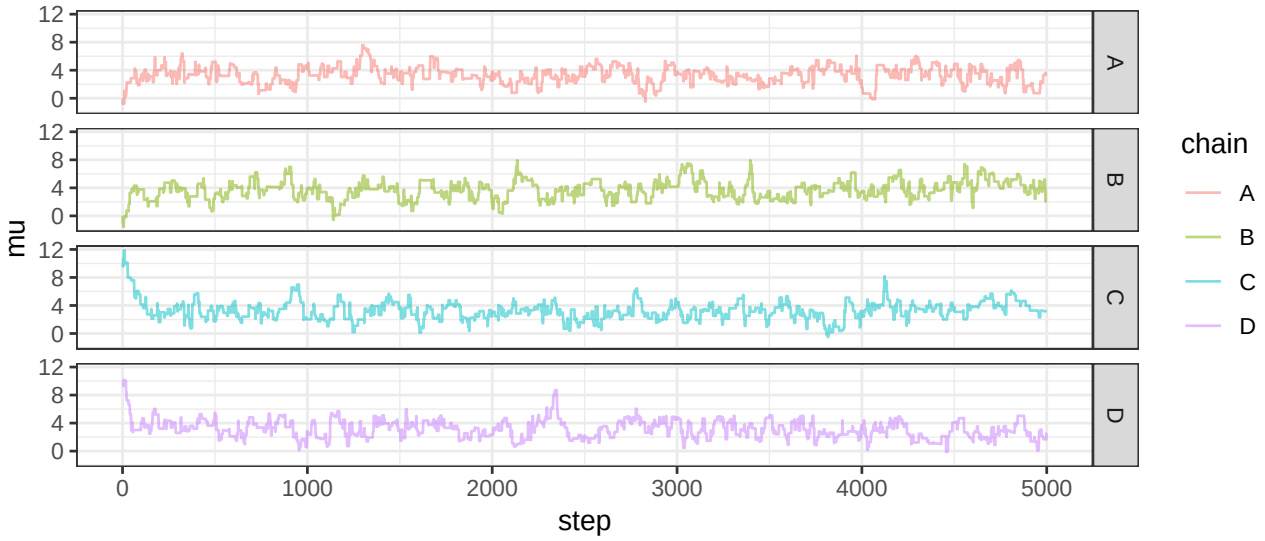
If we run multiple chains with different starting points and different random choices, we hope to see similar trace plots. After all, we don't want our analysis to be an analysis of starting points or of random choices.

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

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



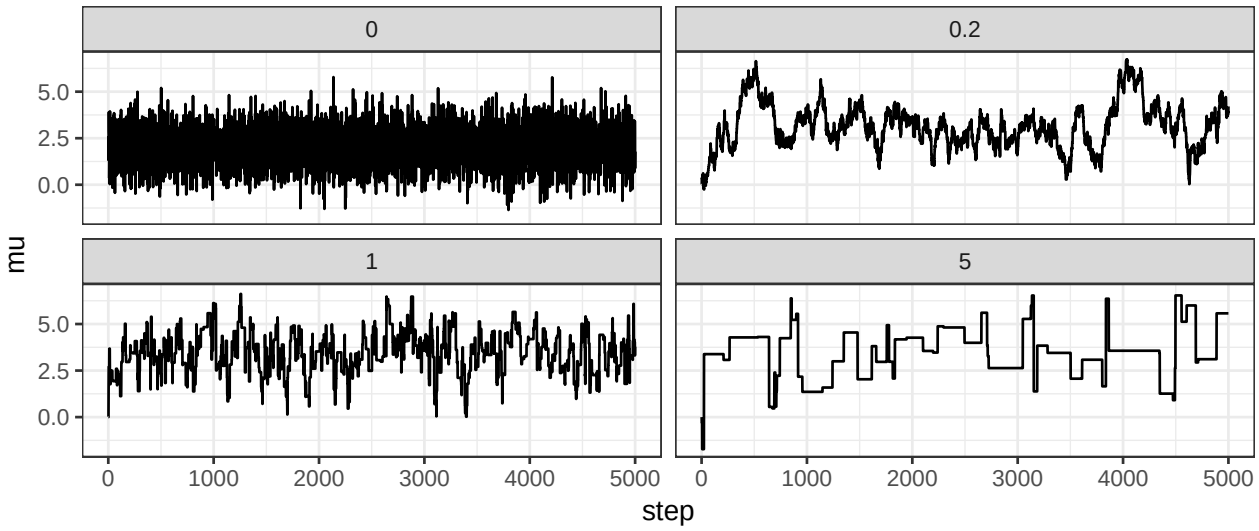

```
gf_line(mu ~ step, color = ~chain, alpha = 0.5, data = Tours1) %>%
  gf_facet_grid( chain ~ .)
```



7.5.2.4 Comparing Chains to an Ideal Chain

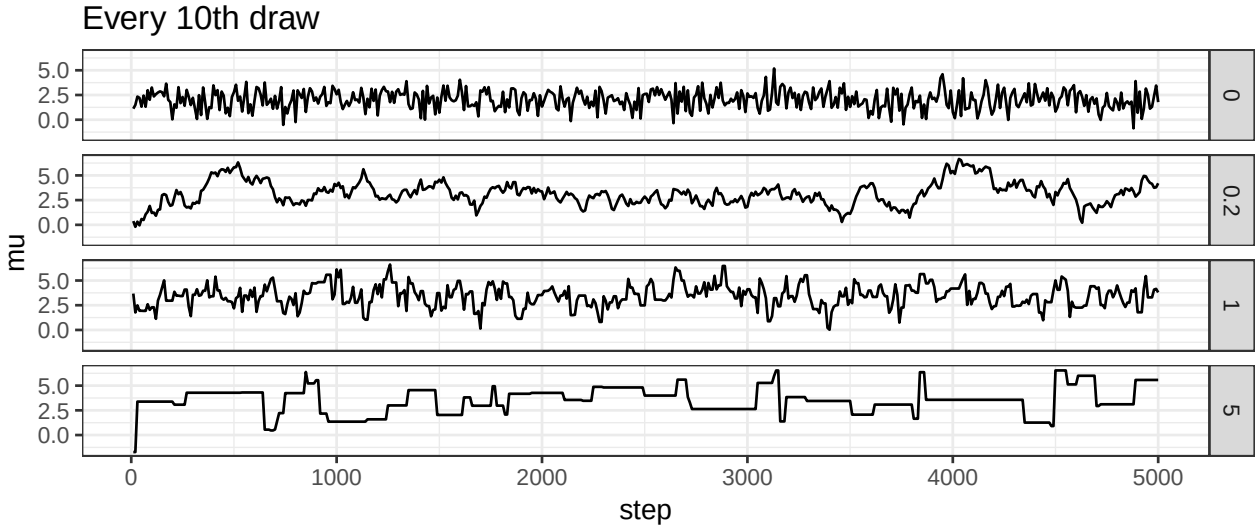
Not all posteriors are normal, but here's what a chain would look like if the posterior is normal and there is no correlation between draws.

```
Ideal <- tibble(step = 1:5000, mu = rnorm(5000, 2, 1), size = 0)
gf_line(mu ~ step | size, data = Tours %>% bind_rows(Ideal))
```

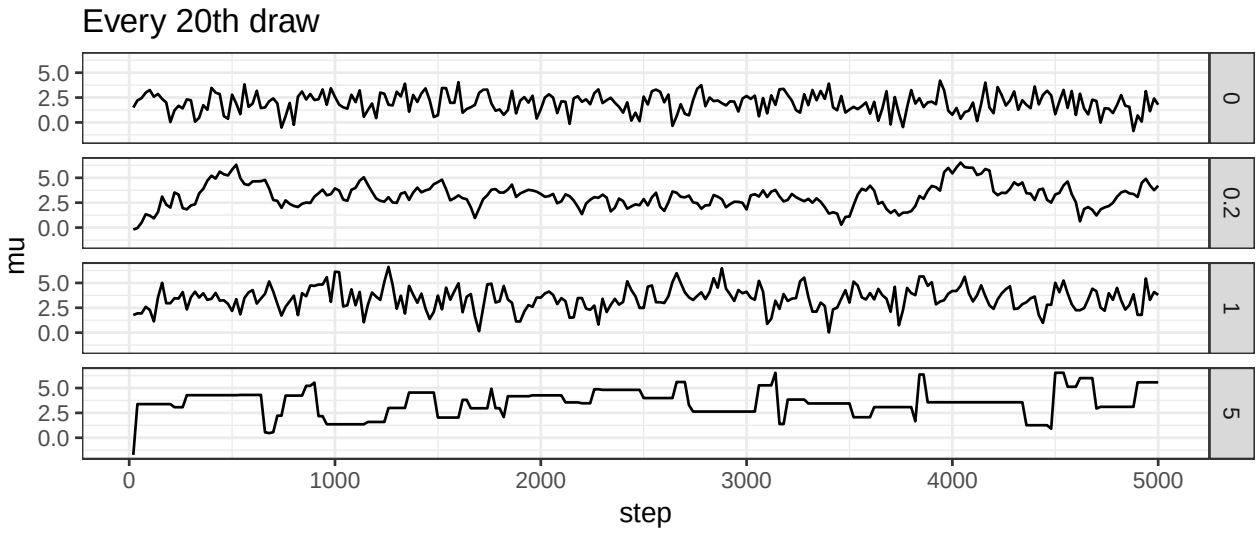


If the draws are correlated, then we might get more ideal behavior if we selected only a subset – every 10th or every 20th value, for example. This is the idea behind "effective sample size". The effective sample size of a correlated chain is the length of an ideal chain that contains as much independent information as the correlated chain.

```
gf_line(mu ~ step | size ~ ., data = bind_rows(Tours, Ideal) %>% filter(step %% 10 == 0)) %>%
  gf_labs(title = "Every 10th draw")
```




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

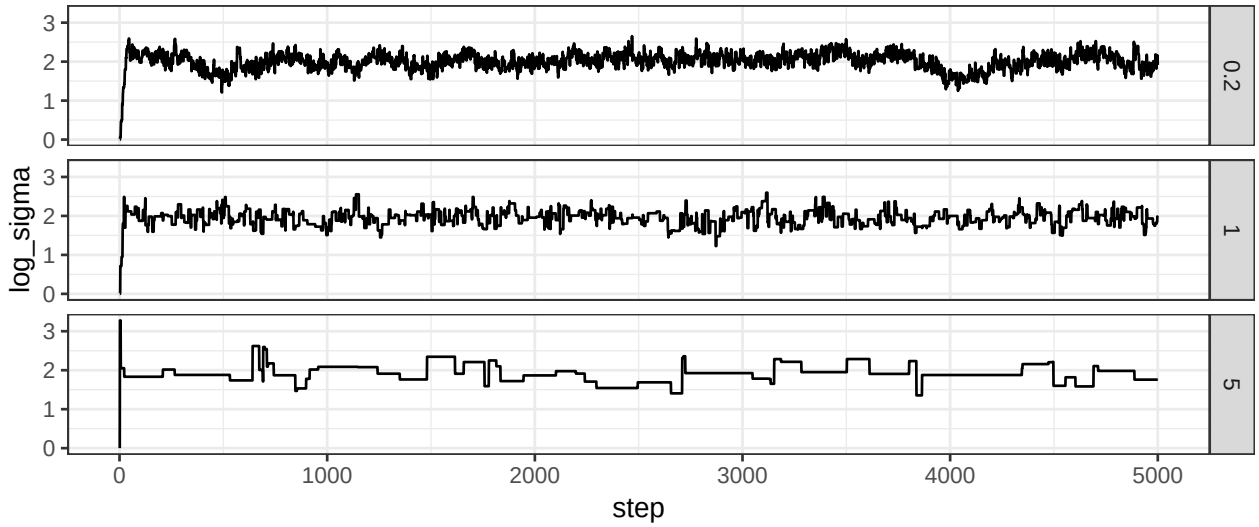


If we thin to every 20th value, our chain with `size = 1` is looking quite similar to the ideal chain. The chain with `size = 0.2` still moves to slowly from place to place, and the chain with `size = 5` is still getting stuck in one place too long. So tuning paramters will affect the effective sample size.

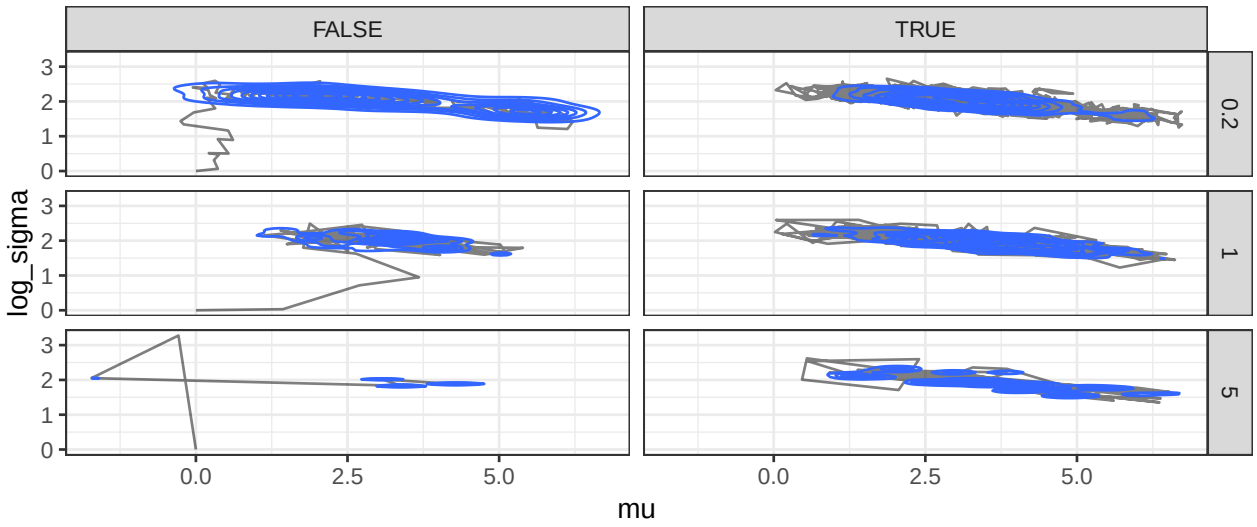
7.5.2.5 Discarding the first portion of a chain

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.

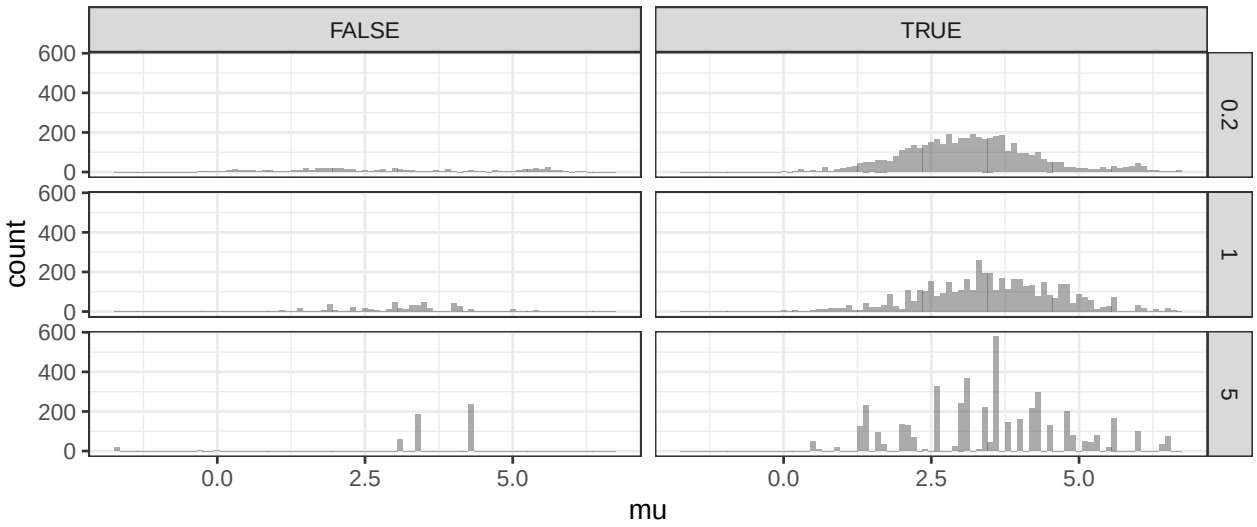
```
gf_line(log_sigma ~ step | size ~ ., data = Tours)
```



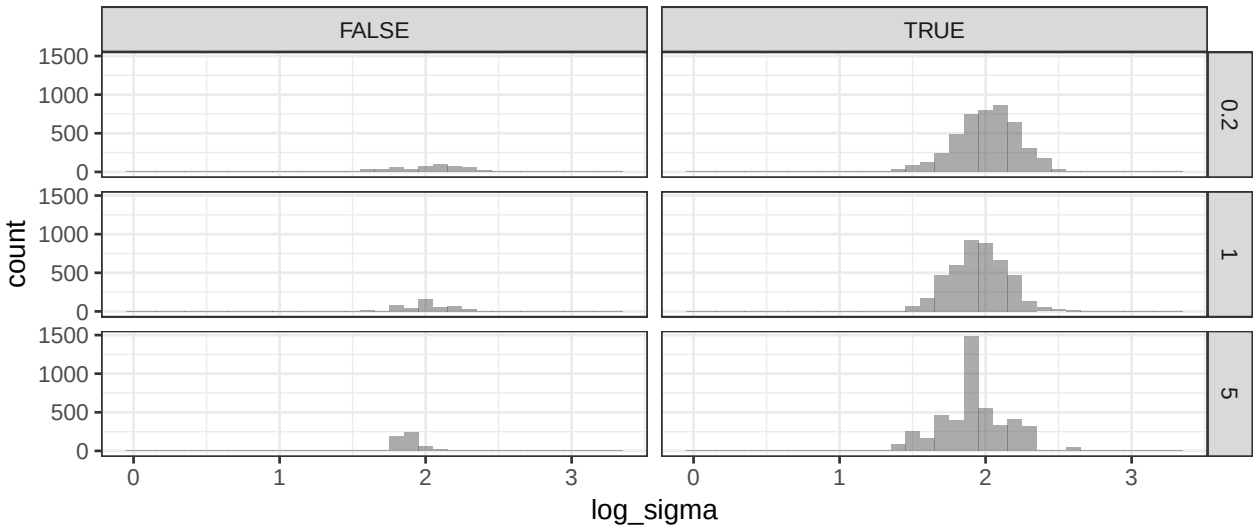

```
gf_path(log_sigma ~ mu | size ~ (step > 500), data = Tours, alpha = 0.5) %>%
  gf_density2d(log_sigma ~ mu, data = Tours)
```




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




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

As it turns out, we won't use the Metropolis or Metropolis-Hastings algorithms in practice. Insead we will use Stan, which shares many features in common with the Metropolis algorithm but has demonsrated that it works well in a wide variety (but not all) models with better performance. Performance improvements are especially notable when the number of parameters in the model is large.

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 intersting models soon.

8.1 What JAGS is

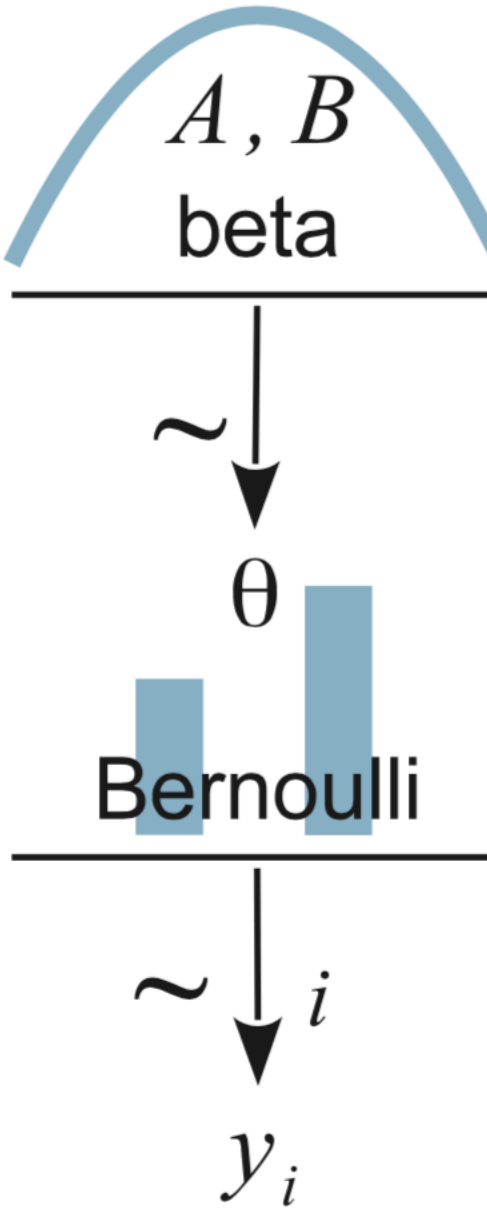
JAGS (Just Another Gibbs Sampler) is an implementation of an MCMC algorithm called Gibbs sampling to sample the posterior distribution of a Bayesian model.

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

- R2jags – interface between R and JAGS
- coda – general tools for analysing and graphing MCMC algorithms
- bayesplot – a number of useful plots using `ggplot2`
- CalvinBayes – includes some of the functions from Kruschke's text

8.2 A Complete Example: estimating a proportion

8.2.1 The Model



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

We can specify the model by creating a text file containing the JAGS description of the model or by creating a special kind of function. The avoids creating temporary files and keeps tidy in our R markdown documents. 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.

```
bern_model <- function() {
  for (i in 1:N) {
    # each response is Bernoulli with fixed parameter theta
    y[i] ~ dbern(theta)
  }
  theta ~ dbeta(1, 1)    # prior for 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).

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//RtmpLMBKUq/model180de2b5
##  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).
```

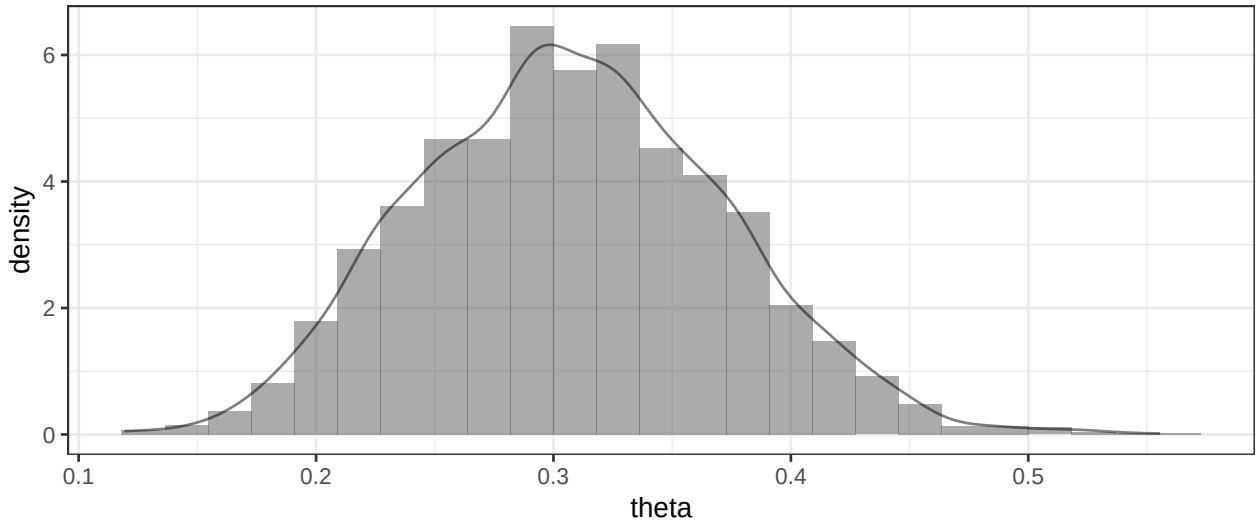
We see that the average value of `theta` in our posterior distribution is 0.3075. The values of `Rhat` and `n.eff` give a quick check that nothing disasterous seems to have happened when we fit this simple model. (`Rhat` should be very nearly 1 if the MCMC algorithm has converged. Because `n.eff` is close to $3000 = 3 \cdot 1000$, this model is sampling the posterior very effeciently.)

We can plot the posterior distribution.

```
library(CalvinBayes)
head(posterior(bern_jags))
```

deviance	theta
61.85	0.2455
61.13	0.3129
61.13	0.2860
61.13	0.3133
61.83	0.3577
62.85	0.2193

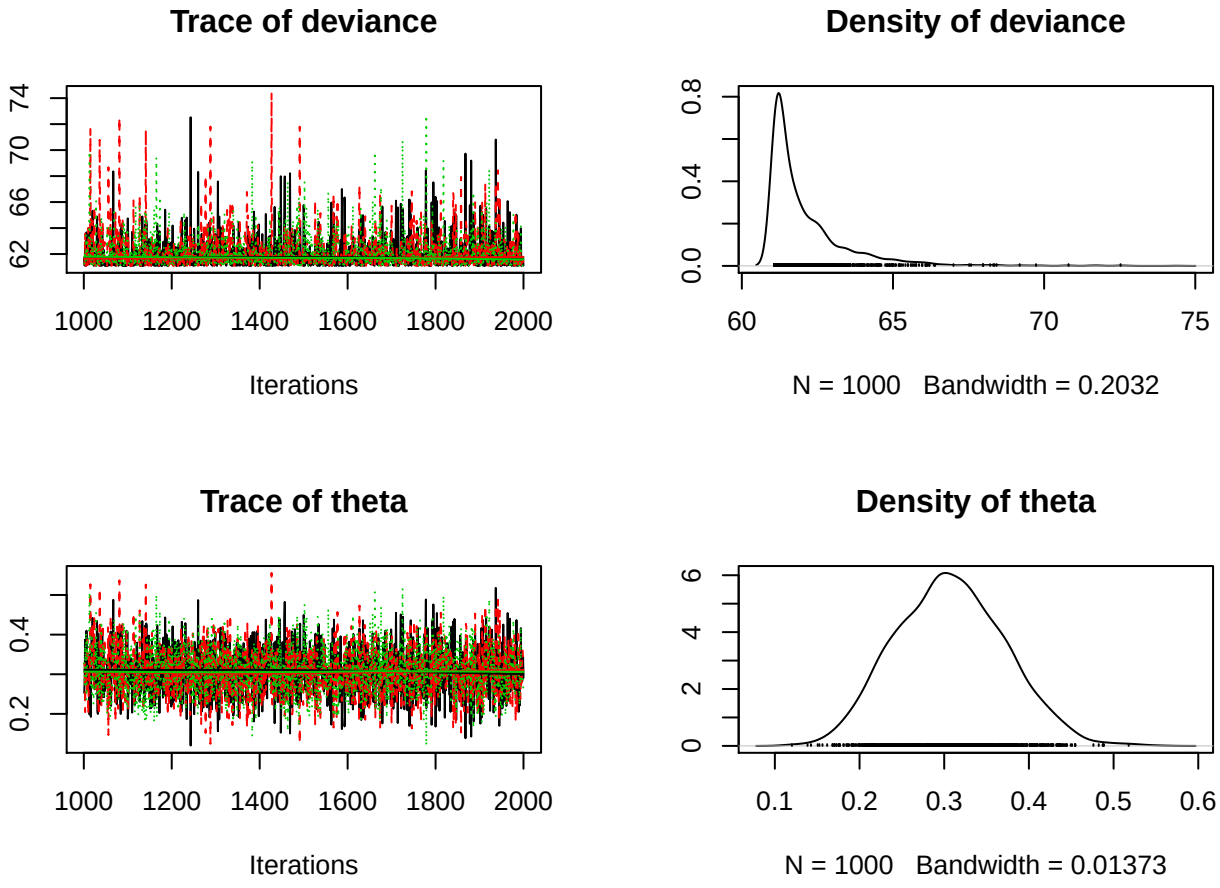
```
gf_dhistogram(~theta, data = posterior(bern_jags)) %>%
gf_dens(~theta, data = posterior(bern_jags))
```



8.2.5 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)
```

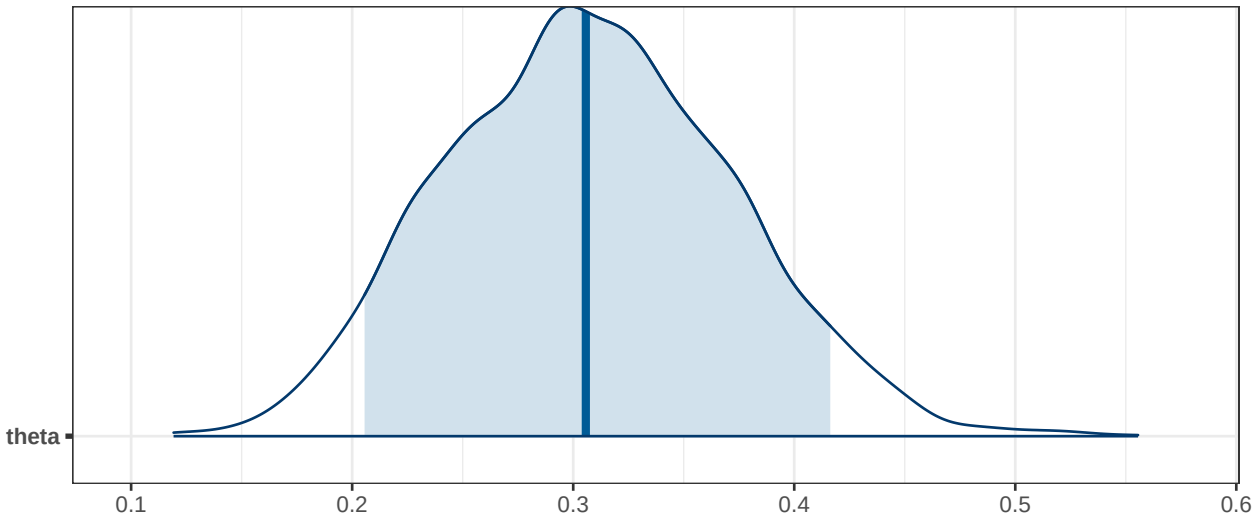


Note: Kruschke uses `rjags` without `R2jags`, so he does this step using `rjags::coda.samples()` instead of `as.mcmc()`. Both functions result in the same thing – posterior samples in a format that `coda` expects, but they have different starting points.

8.2.6 Using bayesplot

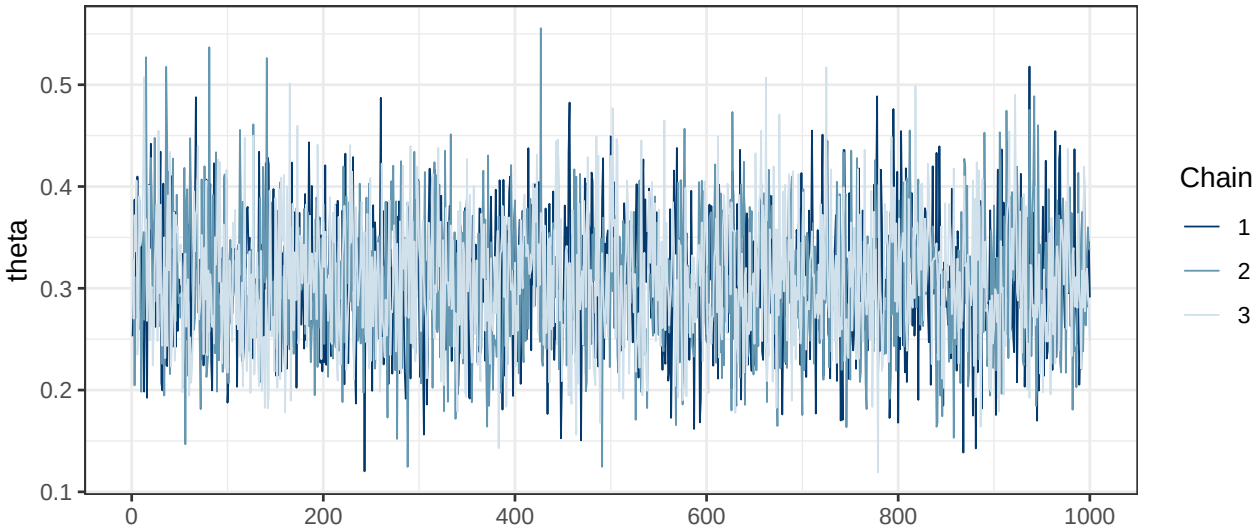
The mcmc object we extracted with `as.mcmc()` can be used by the utilities in the `bayesplot()`. Here, for example is the `bayesplot` plot of the posterior distribution for theta. By default, a vertical line segment is drawn at the median of the posterior distribution.

```
library(bayesplot)
mcmc_areas(
  bern_mcmc,
  pars = c("theta"),      # make a plot for the theta parameter
  prob = 0.90)            # shade the central 90%
```



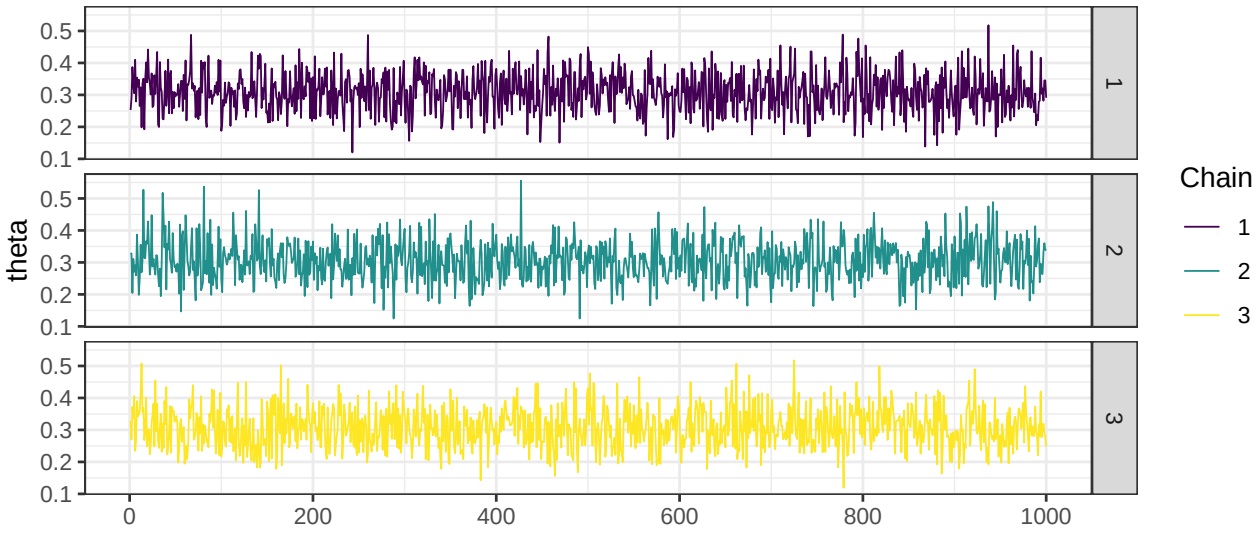
One advantage of `bayesplot` is that the plots use the `ggplot2` system and so interoperate well with `ggformula`.

```
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.
```



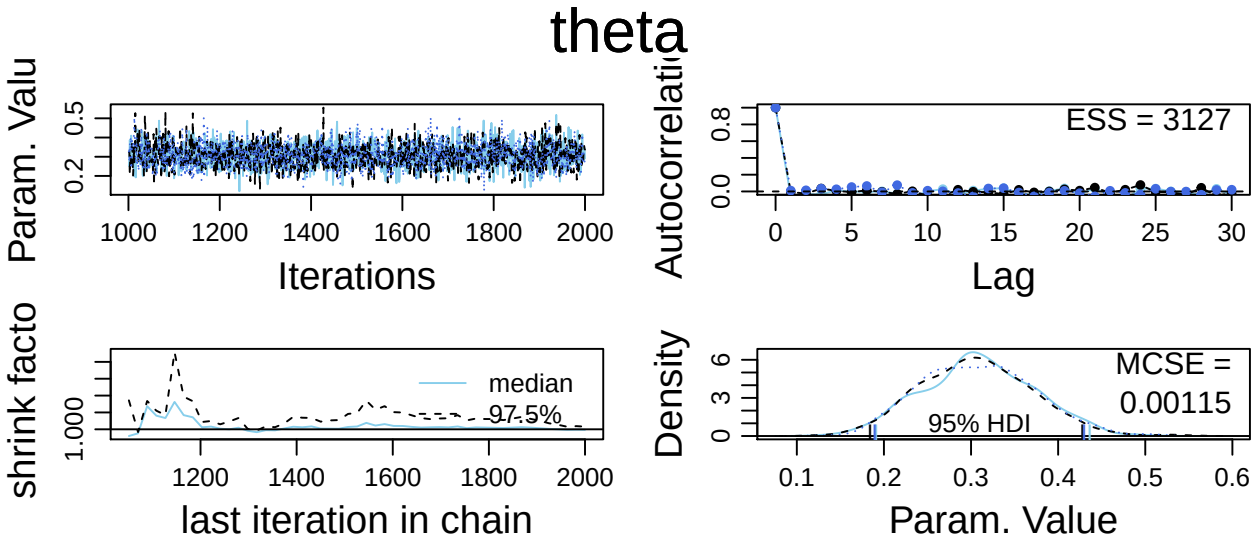
We will encounter additional plots from `bayesplot` as we go along.

8.2.7 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.

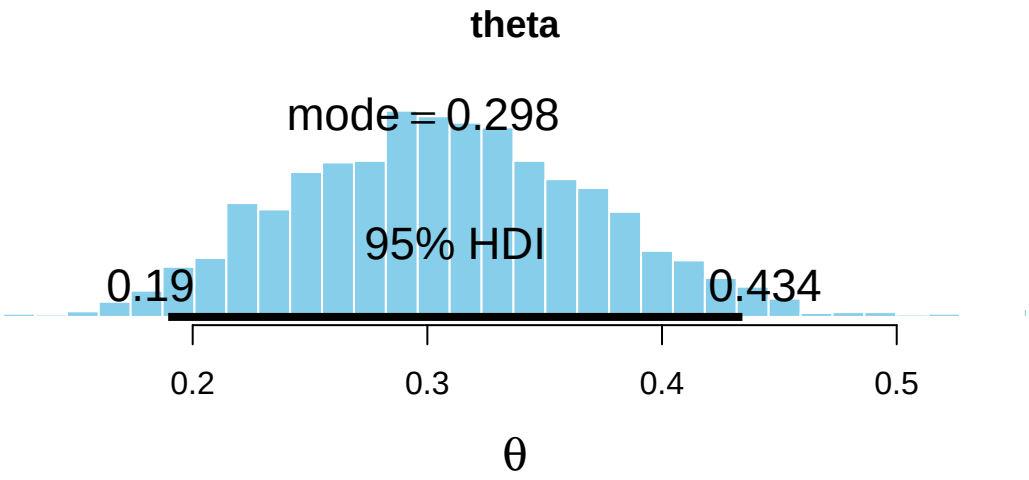
8.2.7.1 diagMCMC()

```
diagMCMC(bern_mcmc, par = "theta")
```



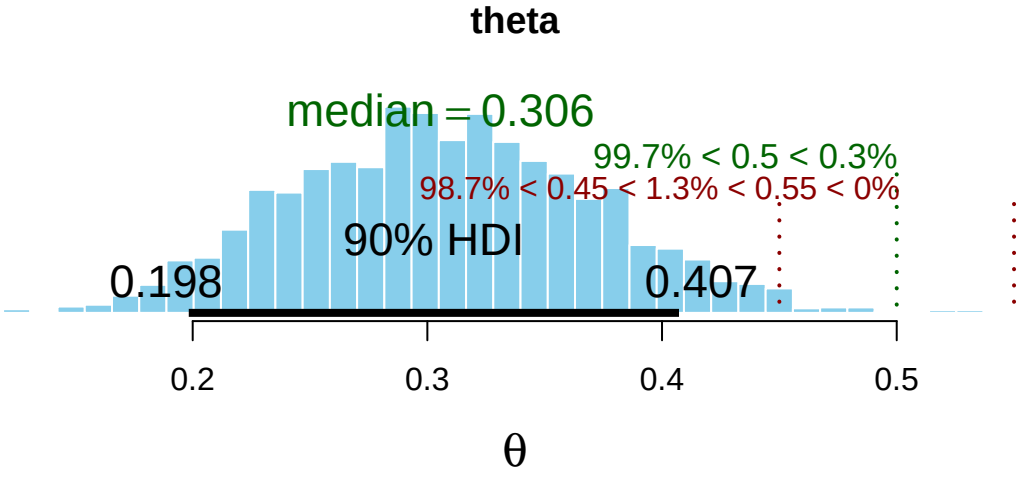
8.2.7.2 plotPost()

```
plotPost(bern_mcmc[, "theta"], main = "theta", xlab = expression(theta))
```



	ESS	mean	median	mode	hdiMass	hdiLow	hdiHigh	compVal	pGtCompVal	ROPElow	ROPEh
theta	3000	0.3075	0.3056	0.2982	0.95	0.1896	0.4342	NA	NA	NA	

```
plotPost(bern_mcmc[, "theta"], main = "theta", xlab = expression(theta),
         cenTend = "median", compVal = 0.5, ROPE = c(0.45, 0.55),
         credMass = 0.90)
```



	ESS	mean	median	mode	hdiMass	hdiLow	hdiHigh	compVal	pGtCompVal	ROPElow	ROPEh
theta	3000	0.3075	0.3056	0.2982	0.9	0.1983	0.4072	0.5	0.0033	0.45	0

8.2.8 Optional arguments to jags()

8.2.8.1 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.burning`: number of burn in steps per chain

```
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,
  )
```

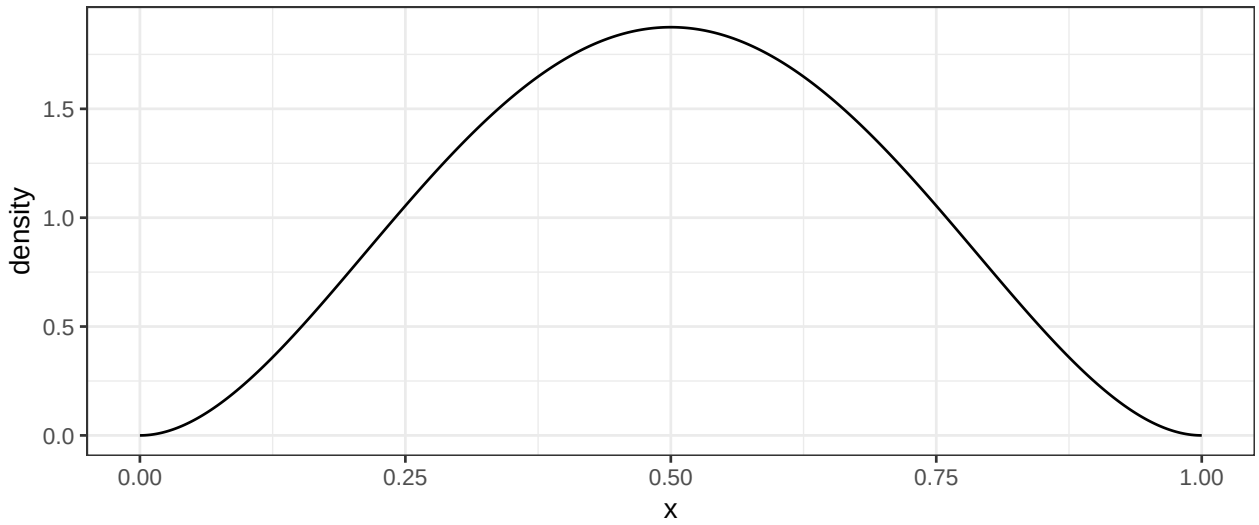
```
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 50
##    Unobserved stochastic nodes: 1
##    Total graph size: 53
##
## Initializing model
```

8.2.8.2 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"),
    n.chains = 4, n.iter = 5000, n.burnin = 1000,
    # start each chain "near" 0.5
    inits = function() list(theta = rbeta(1, 3, 3))
  )
```

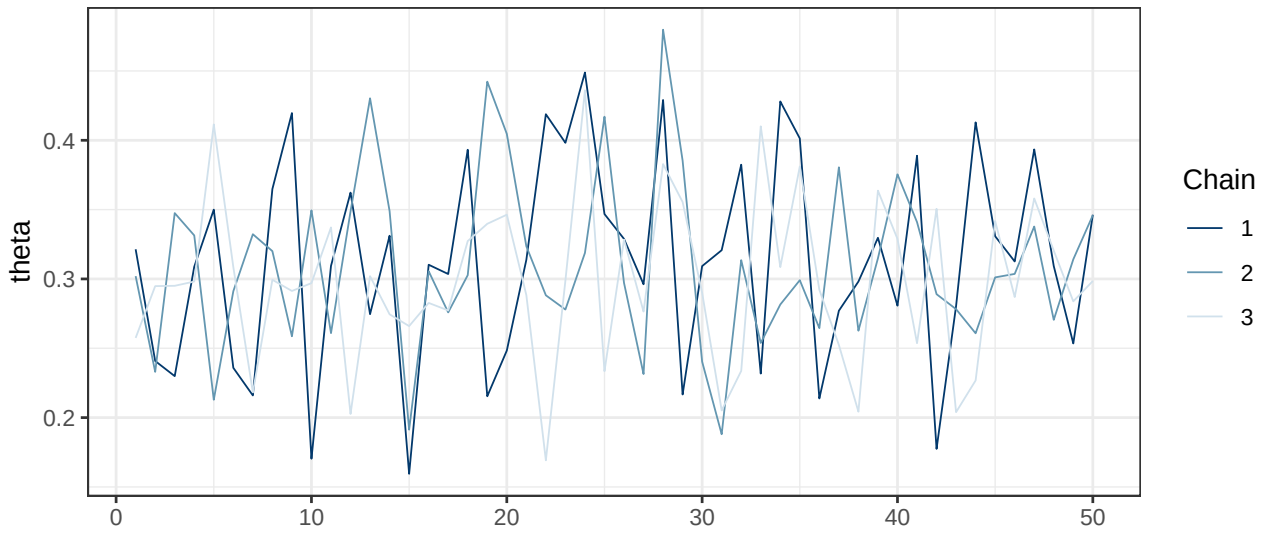
```
## Compiling model graph
```

```
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 50
##    Unobserved stochastic nodes: 1
##    Total graph size: 53
##
## Initializing model
```

```
bern_jags4 <-
  jags(
    data = list(y = z15N50$y, N = nrow(z15N50)),
    model.file = bern_model,
    parameters.to.save = c("theta"),
    n.chains = 3, n.iter = 550, n.burnin = 500,
    # choose specific starting point for each chain
    inits = list(
      list(theta = 0.5), list(theta = 0.3), list(theta = 0.7)
    )
  )
```

```
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 50
##    Unobserved stochastic nodes: 1
##    Total graph size: 53
##
## Initializing model
```

```
mcmc_trace(as.mcmc(bern_jags4), pars = "theta")
```



8.2.8.3 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_jags3 <-
  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
  )
```

8.3 Example 2: comparing two proportions

8.3.1 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 attampt (the subject) and whether the attempt succeeded (0 or 1).

```
library(mosaic)
head(z6N8z2N7)
```

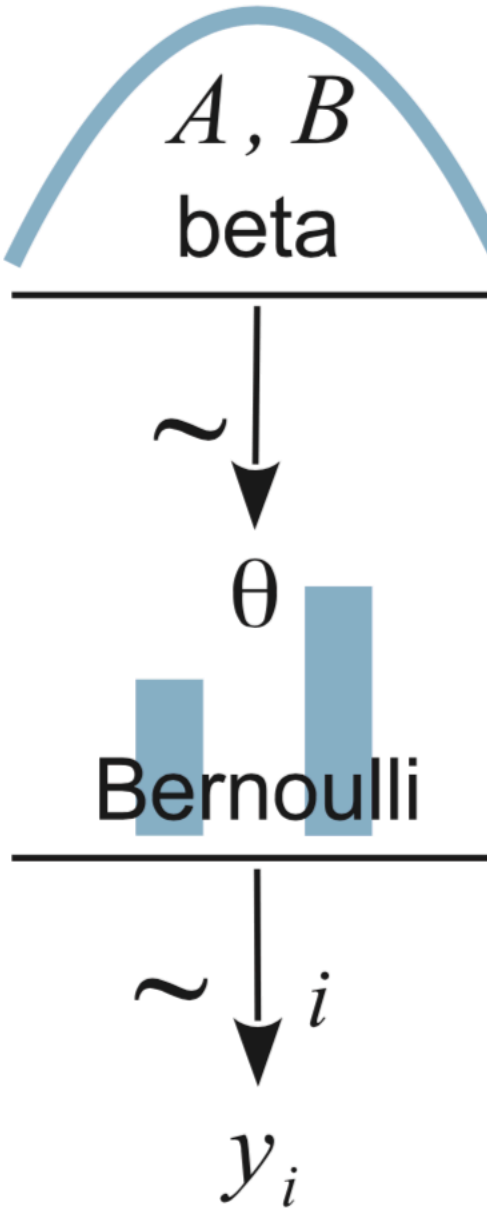
y	s
1	Reginald
0	Reginald
1	Reginald
1	Reginald
1	Reginald
1	Reginald

```
# 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

8.3.2 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

```
hit ~ dbern(theta[subject[i]])
```

where `subject[i]` tells us which subject the `i`th observation was for.

8.3.3 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 JAGSS.)
- `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
```

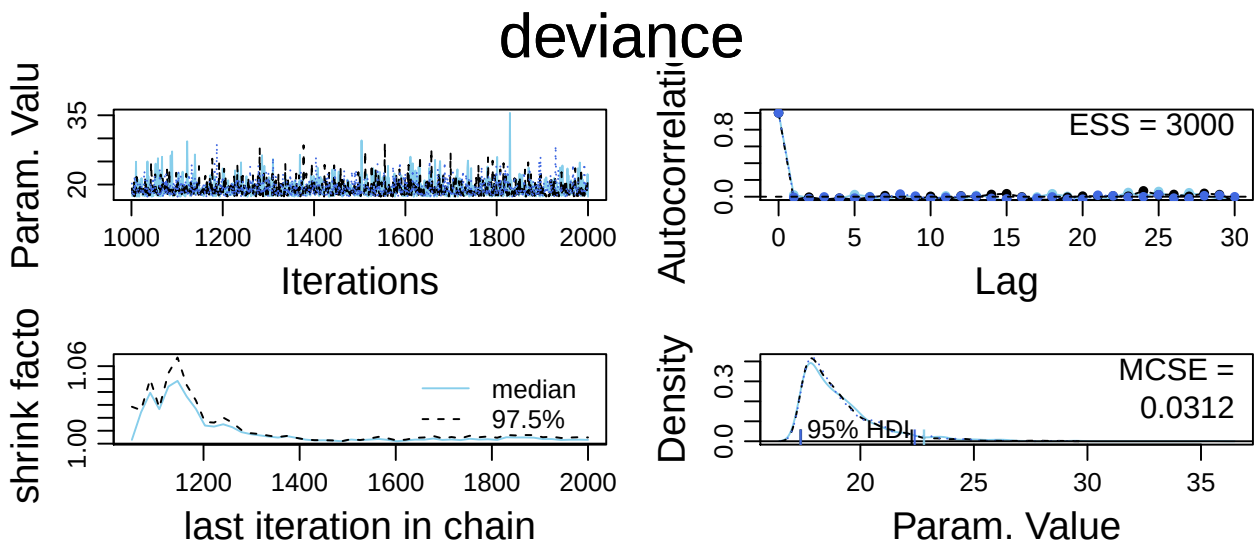
8.3.4 Fitting the model

```
bern2_jags <-
  jags(
    data = TargetList,
    model.file = bern2_model,
    parameters.to.save = "theta")
```

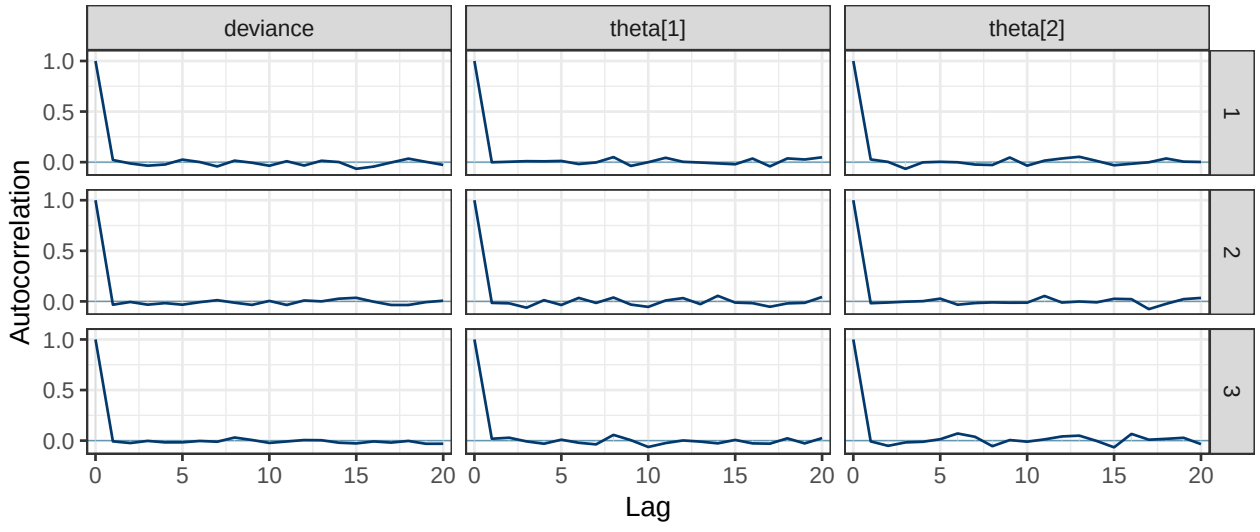
```
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 15
##    Unobserved stochastic nodes: 2
##    Total graph size: 35
##
## Initializing model
```

8.3.5 Inspecting the results

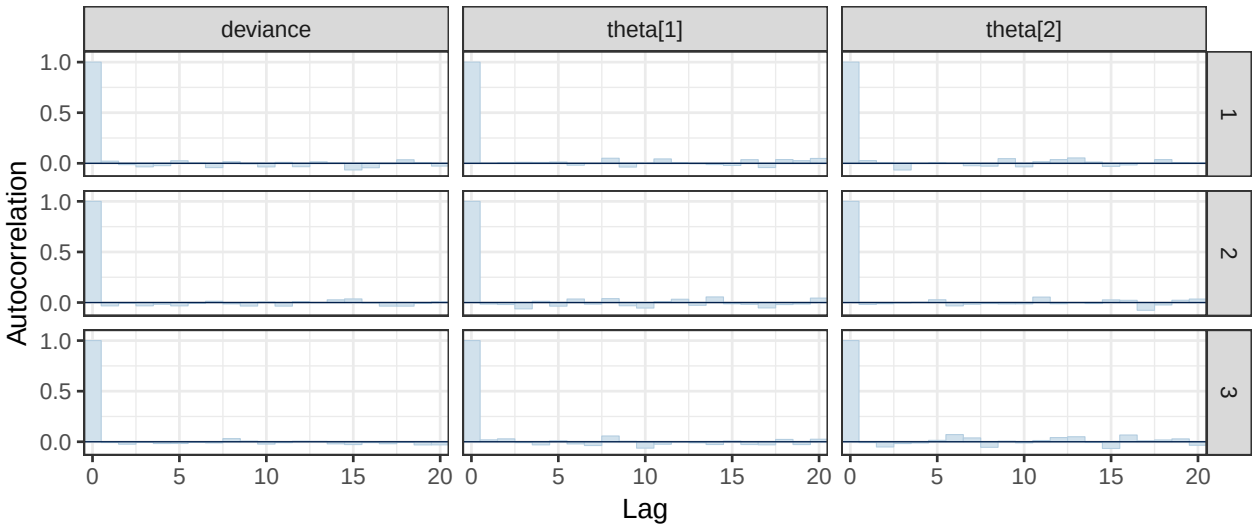
```
bern2_mcmc <- as.mcmc(bern2_jags)
diagMCMC(bern2_mcmc)
```



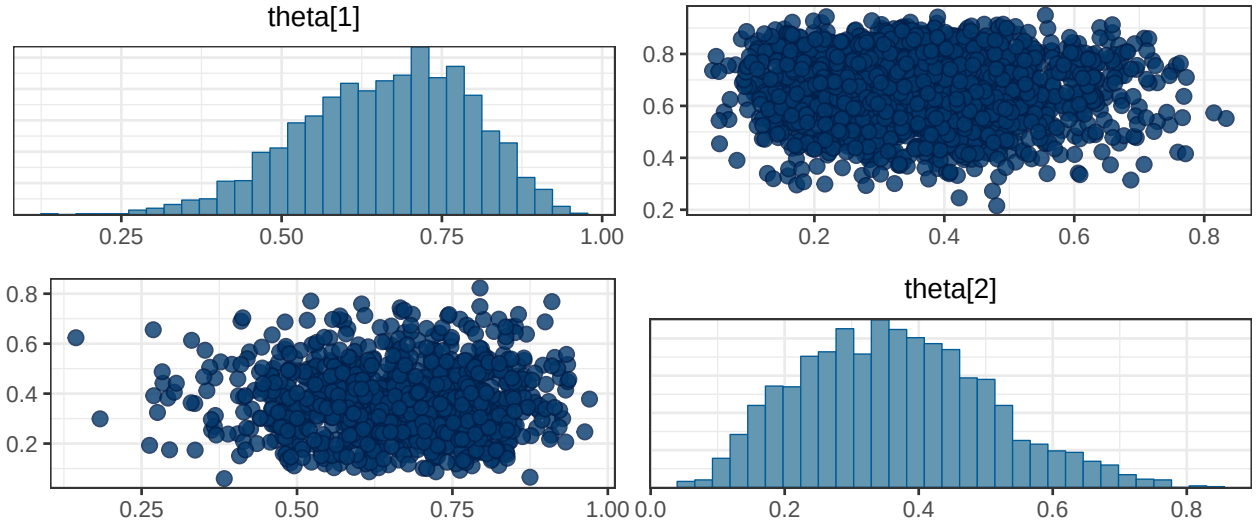

```
# bayesplot plots
mcmc_acf(bern2_mcmc)
```



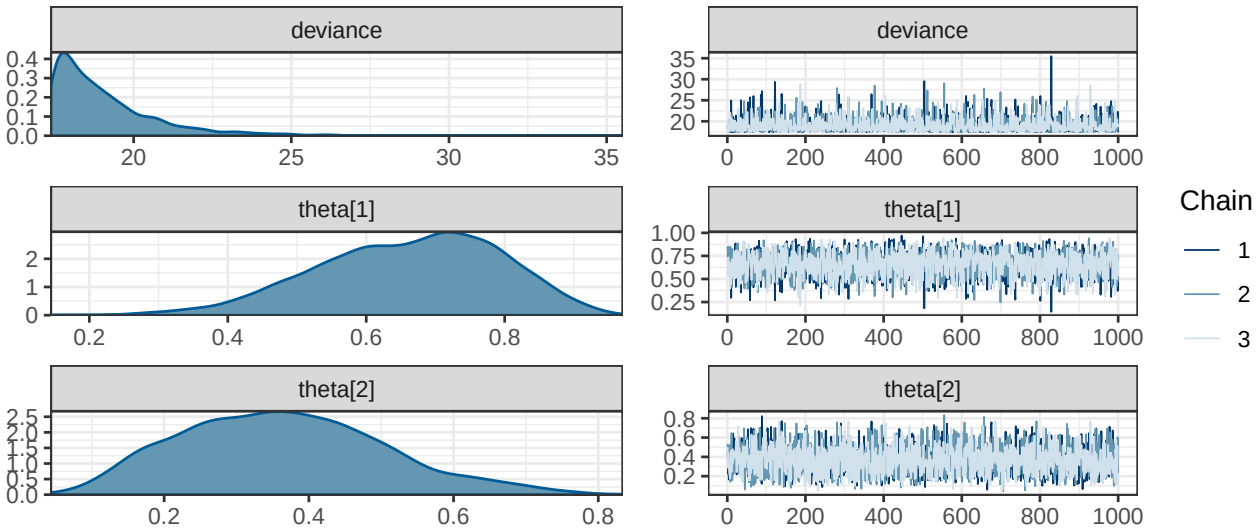

```
mcmc_acf_bar(bern2_mcmc)
```



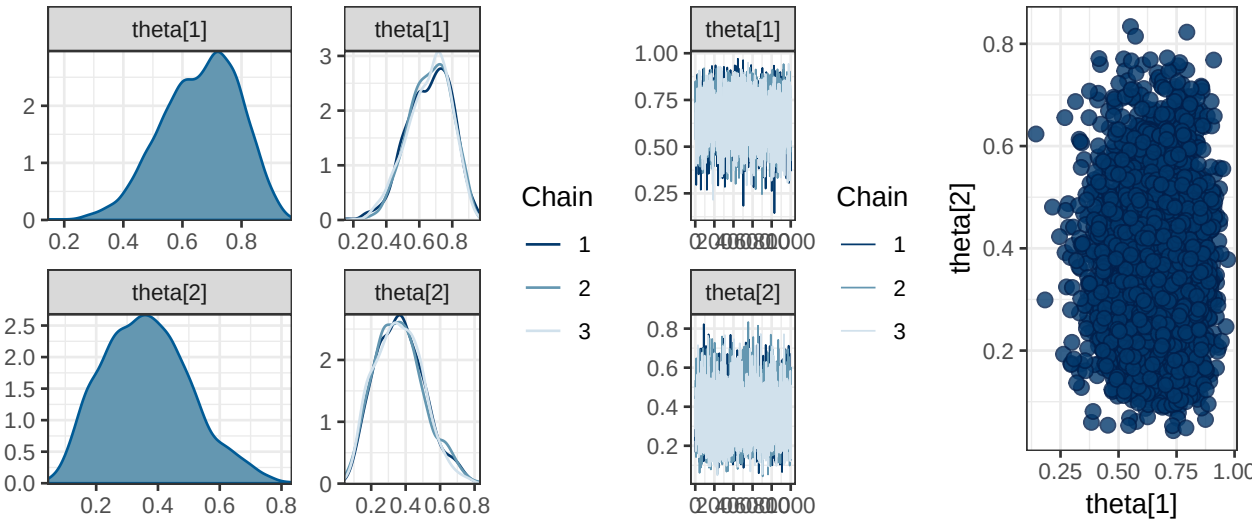

```
mcmc_pairs(bern2_mcmc, pars = c("theta[1]", "theta[2]"))
```




```
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 infromation in a different way or if you just want to inspect the data to make sure you understand the plot.

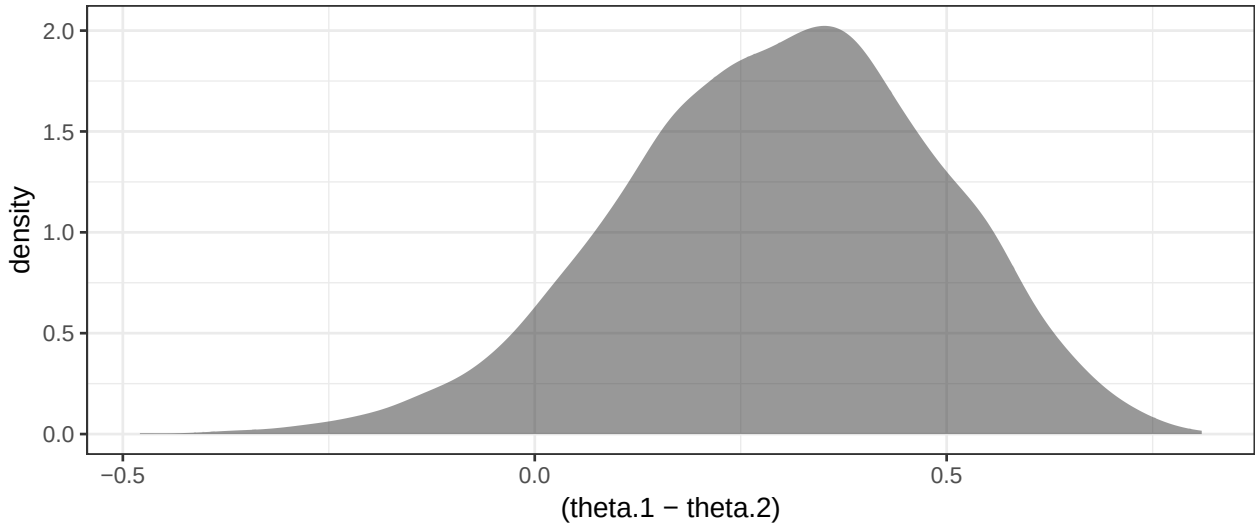
8.3.6 Differnce in proportions

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
18.54	0.5708	0.2464
23.64	0.3772	0.5269
18.69	0.6122	0.4316
18.98	0.5956	0.1552
19.26	0.5277	0.2124
19.27	0.5204	0.3450

```
gf_density( ~(theta.1 - theta.2), data = posterior(bern2_jags))
```



8.3.7 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 witout 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"),
    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
```

8.3.7.1 Note about : in JAGS and in R

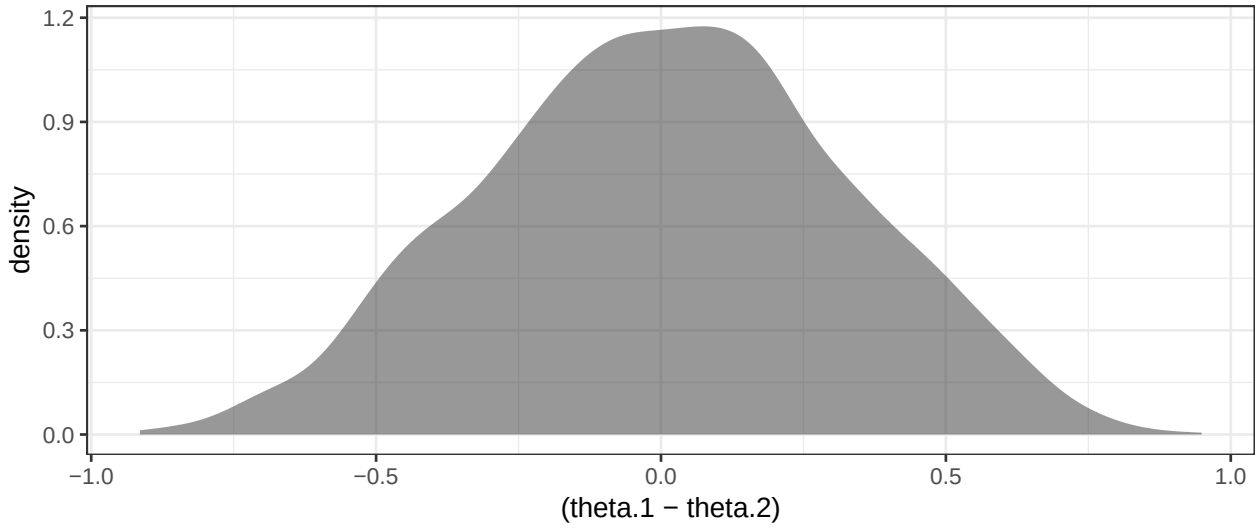
From the JAGS documentation: > The sequence operator : can only produce increasing sequences. If n < m then `m:n` produces a vector of length zero and when this is used in a for loop index expression the contents of loop inside the curly brackets are skipped. Note that this behaviour is different from 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).

8.3.7.2 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.

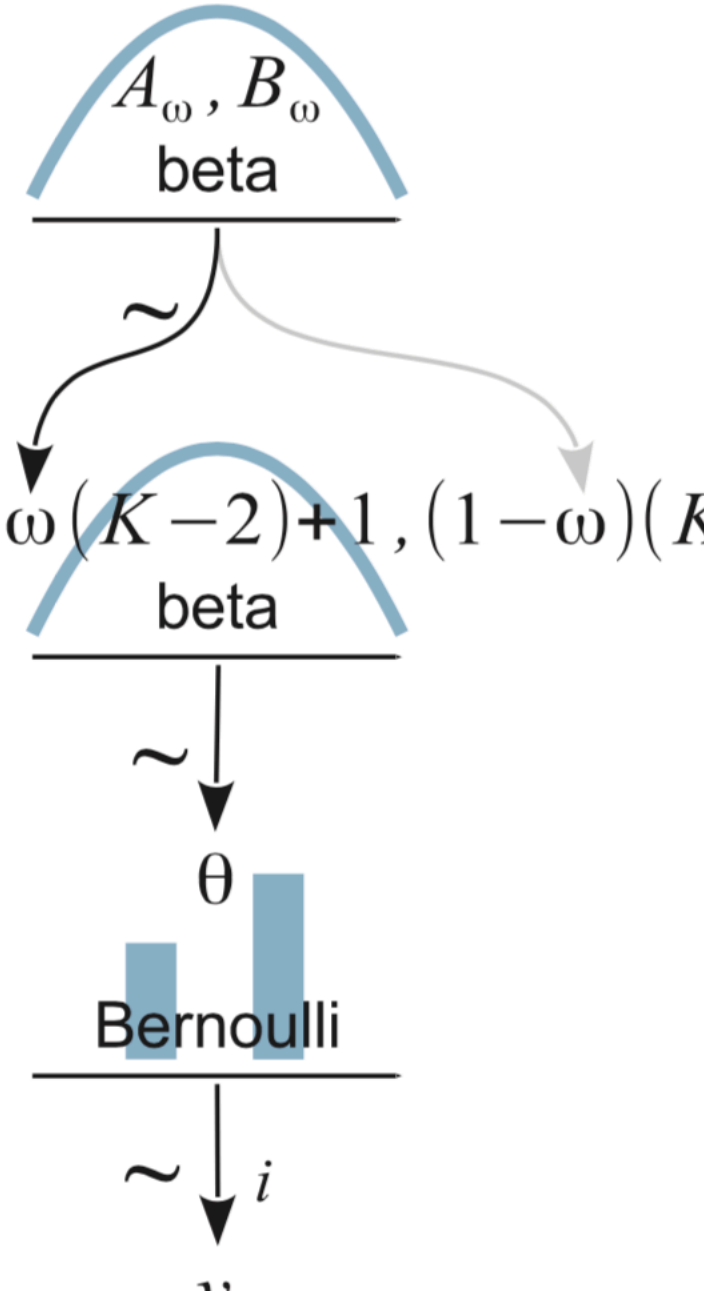
```
gf_density( ~(theta.1 - theta.2), data = posterior(bern2_jags0))
```



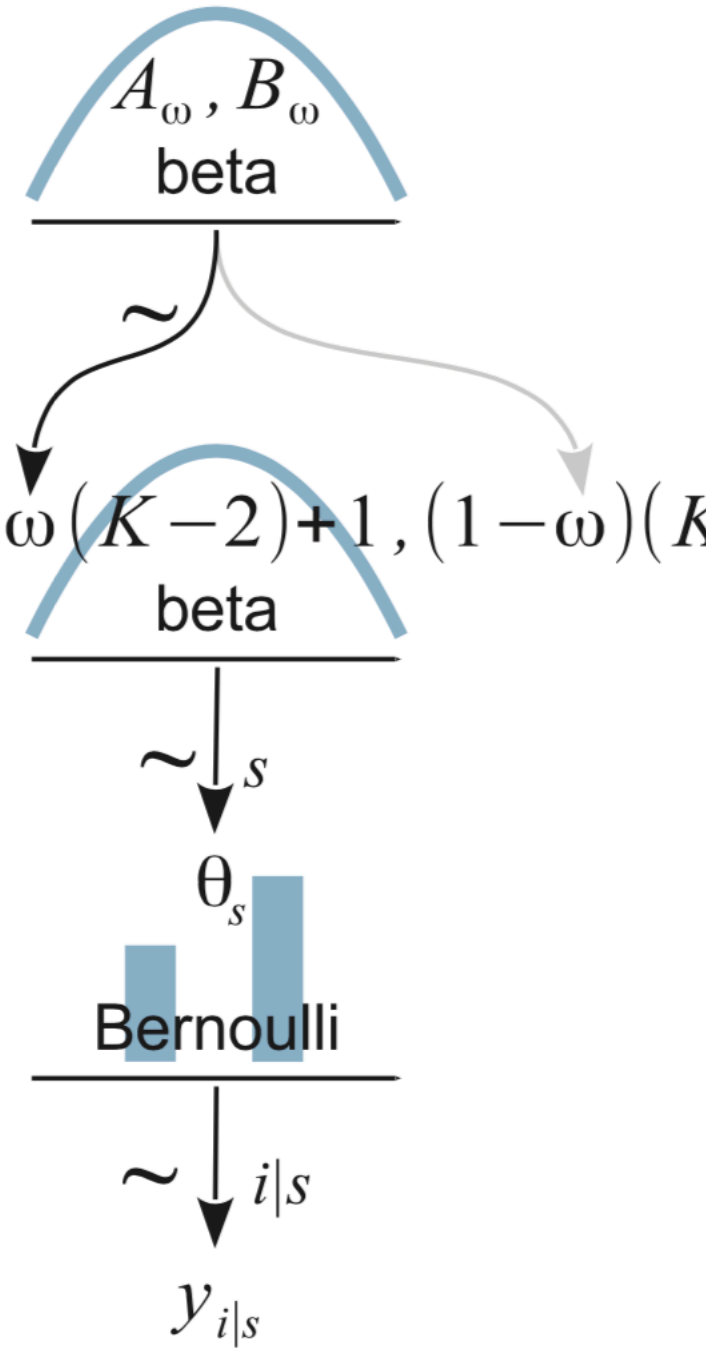
Chapter 9

Heierarchical Models

9.1 One coin from one mint



9.2 Multiple coins from one mint



9.3 Multiple coins from multiple mints