Bayesian Estimation of Regression Models

An Appendix to Fox & Weisberg *An R Companion to Applied Regression*, third edition

John Fox

last revision: 2018-10-01

Abstract

In this appendix to Fox and Weisberg (2019), we review the basics of Bayesian estimation and show how to use the Stan program, via the **rstan** package, for Bayesian estimation of regression models in R.

There are many R packages, some of them self-contained, for Bayesian estimation of statistical models. Other packages interface R with independent software for Bayesian estimation. In this appendix, we illustrate the use of the **rstan** package (Stan Development Team, 2018), which provides an R interface to the Stan program (Carpenter et al., 2017). Stan represents the state of the art in Bayesian statistical software, permitting the flexible definition of probability models for data, automatically compiling the models into C++ code and then into independently executable programs, and estimating the models by Hamiltonian Monte Carlo.

Stan is included with the **rstan** package, but for it to work it's necessary to have a C++ compiler installed on your computer; we explain how to do this on Windows and macOS systems in the Appendix to this appendix.

Bayesian inference for regression models and its implementation in R is a very large topic worthy of book-length treatment, for which we refer you to the sources in the complementary readings at the end of this appendix. The appendix simply sketches the topic.

1 Basic Ideas

To review the basics of Bayesian estimation, we adapt an example from Fox (2009, Sec. 3.7), which is coincidentally similar to an example in Carpenter et al. (2017, Sec. 2.1): Imagine that we wish to estimate the probability $0 \leq \phi \leq 1$ that a coin comes up heads, thinking of this unknown parameter as a property of the coin. We're impatient, so we collect data by flipping the coin just 10 times, producing the sequence $\{H, H, T, H, T, T, T, T, H, T\}$ of heads and tails. The probability of observing this sample of data given the parameter ϕ is

$$\begin{aligned} \Pr(\text{data}|\text{parameter}) &= \Pr(HHTHTTTTHT|\phi) \\ &= \phi\phi(1-\phi)\phi(1-\phi)(1-\phi)(1-\phi)\phi(1-\phi) \\ &= \phi^4(1-\phi)^6 \end{aligned}$$

which is the probability of 10 independent Bernoulli trials $Y_i, i = 1, \ldots, 10$, where $Y_i = 1$ when the ith flip produces a head, zero when it produces a tail, and for which $\Pr(Y_i = 1) = \phi$.

The *likelihood function* turns this equation around by treating the observed data as conditionally fixed and the unknown parameter ϕ as variable over its potential range:

$$\begin{aligned} L(\text{parameter}|\text{data}) &= \Pr(\phi|HHTHTTTTHT) \\ &= \phi^4(1-\phi)^6 \end{aligned}$$

The probability ϕ is continuously variable between zero and 1; here are some representative values of the likelihood over that range:[1]

ϕ	$L(\phi)$
0.0	0
0.1	0.00005314
0.2	0.00041943
0.3	0.00095296
0.4	0.00119439
0.5	0.00097656
0.6	0.00053084
0.7	0.00017503
0.8	0.00002621
0.9	0.00000066
1.0	0

As is sensible, $L(0) = 0$, because if ϕ were zero, then we could observe no heads, and we observed 4 heads; similarly, $L(1) = 0$, because if $\phi = 1$, then we could observe no tails. Figure 1, drawn by the following R commands, shows the full likelihood function:[2]

```
phi <- seq(0, 1, length=500)
L <- phi^4 * (1 - phi)^6
plot(phi, L, type="l", lwd=2, xlab=expression(phi), ylab=expression(L(phi)))
abline(h=0, col="darkgray")
abline(v=0.4, h=max(L), lty=2)
text(0.4, -0.00025, expression(hat(phi)), xpd=TRUE)
```

The maximum of the likelihood function is achieved at $\widehat{\phi} = 0.4$, the *maximum-likelihood estimate* of ϕ, which is just the observed proportion 4/10 of heads.

In contrast to maximum-likelihood estimation, which is based solely on the observed data, *Bayesian inference*[3] requires that we express our prior beliefs about ϕ as a probability density function over its possible values, and combines this *prior distribution* with the likelihood function.

[1] Rather than focusing on the observed sequence of heads and tails, we could instead count the observed number of heads in the 10 independent flips of the coin. That number, say $Y = 4$, has the following probability from the binomial distribution:

$$\Pr(Y = 4) = \frac{10!}{4!(10-4)!}\phi^4(1-\phi)^{10-4}$$

Because $\frac{10!}{4!(10-4)!}$ is a constant, reflecting the fact that the number of heads is a sufficient statistic for ϕ (i.e., it exhausts all information about ϕ in the data), using the binomial likelihood produces the same analysis as using the Bernoulli likelihood.

[2] See Chapter 9 of the *R Companion* for a discussion of customized graphics in R.

[3] Bayesian inference is named after the Reverand Thomas Bayes, an 18th century British mathematician, who formulated Bayes' Theorem,

$$\Pr(A|B) = \frac{\Pr(B|A)\Pr(A)}{\Pr(B)}$$

where the *events* A and B are given the following interpretation:

- A is an uncertain proposition whose truth we want to establish, say that the probability of heads for a coin is 0.5.
- B represents *data* that are relevant to this proposition, such as our observation of a sequence of 4 heads and 6 tails in 10 flips of the coin.
- $\Pr(A)$ is the *prior probability* of A, formulated before we start collecting data and representing the strength of our belief in the truth of A.
- $\Pr(B|A)$ is the *conditional probability* of observing the obtained data assuming A—for example, the probability of the obtained sequence of heads and tails given that the probability of observing a head is 0.5.

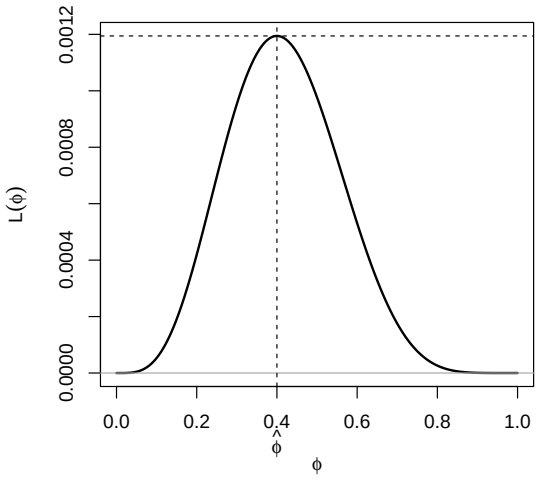

Figure 1: Likelihood function $L(\phi)$ for the coin-flipping experiment, having observed the sequence $\{H, H, T, H, T, T, T, T, H, T\}$ of 4 heads and 6 tails. The maximum-likelihood estimate $\widehat{\phi} = 0.4$ is marked on the graph.

For example, if we believe that all values of ϕ are equally likely then that implies the *flat prior* $p(\phi) = 1$; this is a *proper prior* density because it encloses an area of 1:[4] $\int_0^1 p(\phi)d\phi = 1$.[5] The Bayesian *posterior density* of ϕ is the product of the prior density and the likelihood, normalized to integrate to 1:

$$p(\phi|\text{data}) = \frac{p(\phi)L(\phi|\text{data})}{\int_0^1 p(\phi')L(\phi'|\text{data})d\phi'}$$

For the flat prior $p(\phi) = 1$, the posterior density is simply proportional to the likelihood. If we take as a Bayesian point-estimate of ϕ the mode of the posterior distribution, then the Bayesian and maximum-likelihood estimates coincide. Alternatively, we could use the mean or median of the posterior. As well, we can form a Bayesian analog of a confidence interval, called a *credible interval*, based on quantiles of the posterior distribution; for example, the probability is 0.95 that ϕ is between the 0.025 and 0.975 quantiles. A credible interval therefore has a more straightforward interpretation than a "frequentist" confidence interval: Bayesians regard unknown parameters as random variables,

- $\Pr(B)$ is the *unconditional probability* of B, equal to

$$\Pr(B) = \Pr(B|A)\Pr(A) + \Pr(B|\neg A)[1 - \Pr(A)]$$

  where $\neg A$ ("not A") is the event that the proposition A is false.

- Then $\Pr(A|B)$ is our *posterior probability* of belief in A, updated from the prior probability on the basis of the data we collected.

The analysis that follows in the text uses a version of Bayes' Theorem that applies to continuous random variables rather than to simple events.

[4] A disquieting realization is that a flat prior on ϕ is not in general a flat prior on a transformation of ϕ, such as the logit transformation $\theta = \log[\phi/(1-\phi)]$, and vice-versa. Indeed, a flat prior on θ is *improper* (encloses an infinite area, not 1), because θ is unbounded below and above, and places high probability on values of ϕ that are near zero and 1. It is possible to formulate a prior, called the Jeffreys prior, that is invariant with respect to transformation of a parameter, but that too has its limitations (see, e.g., Gelman et al., 2013, Sec. 2.8).

[5] If you're unfamiliar with calculus, this equation simply states that the area under the horizontal straight line $p(\phi) = 1$ between $\phi = 0$ and $\phi = 1$—that is the unit square—encloses an area of 1.

subject to variation due to uncertainty about their values, not as fixed values.

Suppose, now, that we have stronger prior beliefs about the value of ϕ. For example, we look at the coin and note that it has both a head and a tail, and that it doesn't appear to be strangely weighted. We figure that ϕ must be pretty close to 0.5, almost surely between 0.2 and 0.8. The family of *Beta distributions* provides a flexible set of prior densities for a probability like ϕ. The Beta distributions depend on two *shape parameters*, α and β, and have the density

$$p(\phi) = \frac{\phi^{\alpha-1}(1-\phi)^{\beta-1}}{B(\alpha,\beta)} \quad \text{for} \quad 0 \leq \phi \leq 1$$

where $B(\alpha, \beta)$ is a normalizing constant that makes $p(\phi)$ integrate to 1.[6] Some examples of Beta distributions are shown in Figure 2, with symmetric densities on the left and asymmetric densities, produced by $\alpha \neq \beta$, on the right:

```
.par <- par(mfrow=c(1, 2))
plot(0:1, c(0, 5), type="n", xlab=expression(phi),
     ylab=expression(p(phi)), main="(a)")
lines(phi, dbeta(phi, 1, 1), lwd=2)
lines(phi, dbeta(phi, 4, 4), lwd=2, lty=2)
lines(phi, dbeta(phi, 8, 8), lwd=2, lty=3)
legend("top", inset=0.02, legend=c(expression(alpha == 1 ~~ beta == 1),
                                   expression(alpha == 4 ~~ beta == 4),
                                   expression(alpha == 8 ~~ beta == 8)),
       lty=1:3, lwd=2, title="Symmetric Beta Priors", bty="n")

plot(0:1, c(0, 5), type="n", xlab=expression(phi),
     ylab=expression(p(phi)), main="(b)")
lines(phi, dbeta(phi, 1, 4), lwd=2)
lines(phi, dbeta(phi, 4, 1), lwd=2, lty=2)
lines(phi, dbeta(phi, 4, 8), lwd=2, lty=3)
lines(phi, dbeta(phi, 8, 4), lwd=2, lty=4)
legend("top", inset=0.02, legend=c(expression(alpha == 1 ~~ beta == 4),
                                   expression(alpha == 4 ~~ beta == 1),
                                   expression(alpha == 4 ~~ beta == 8),
                                   expression(alpha == 8 ~~ beta == 4)),
       lty=1:4, lwd=2, title="Asymmetric Beta Priors", bty="n")

par(.par)
```

For our example, therefore, Beta$(8, 8)$ roughly reflects our prior beliefs about ϕ; in contrast, Beta$(1, 1)$ corresponds to the flat (or *noninformative*) prior $p(\phi) = 1$.

Figure 3 shows the alternative posterior densities that we obtain for the flat Beta(1, 1) and *informative* Beta(8, 8) priors. In each case, the posterior is also a Beta distribution, Beta$(5, 7)$ for the flat prior and Beta$(12, 14)$ for the informative prior. That the prior and posterior distributions are in the same (Beta) family reflects the fact that the Beta distribution is a *conjugate prior* for the Bernoulli likeihood.

[6]* The formula for $B(\alpha, \beta)$ isn't terribly important to us, other than to know that it exists. For the record,

$$B(\alpha,\beta) = \frac{\Gamma(\alpha)\Gamma(\beta)}{\Gamma(\alpha+\beta)}$$

where $\Gamma(\nu) = \int_0^\infty e^{-\nu} z^{\nu-1} dz$ is the *gamma function*.

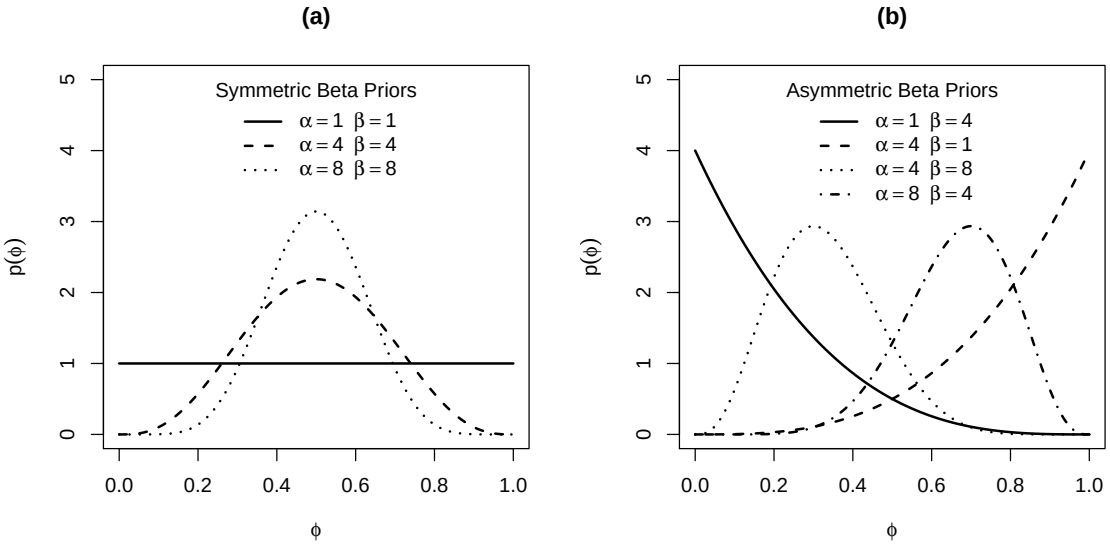

Figure 2: Beta priors for various choices of the shape parameters α and β: (a) symmetric, $\alpha = \beta$; (b) asymmetric, $\alpha \neq \beta$.

```
plot(phi, dbeta(phi, 12, 13), type="l", lwd=2, xlab=expression(phi),
     ylab=expression(p(phi)))
lines(phi, dbeta(phi, 5, 7), lty=2, lwd=2)
legend("topright", legend=c("for Beta(8, 8) prior", "for Beta(1, 1) prior"),
       lty=1:2, lwd=2, title="Posterior Distributions", bty="n", inset=0.02)
```

Using the Beta(8, 8) prior is like adding 7 heads and 7 tails to the data, thus drawing the mode of the posterior distribution much closer to $\phi = 0.5$ and also decreasing the variability of the posterior. Thinking about the Beta prior in this way makes it clear that as the sample size n grows, the impact of the prior on the posterior decreases; eventually, for a large-enough sample, the data (through the likelihood) entirely dominate the prior.

The generalization of this example is immediate: If we flip the coin n times observing h heads and $n - h$ tails in a particular sequence, then the Bernoulli likelihood is $L(\phi) = \phi^h(1-\phi)^{n-h}$. Using the conjugate prior Beta(α, β), the posterior density is Beta$(h + \alpha, n - h + \beta)$, which is like adding $\alpha - 1$ heads and $\beta - 1$ tails to the data.

For our coin-flipping example, we formulated a probability model with a single parameter, ϕ, the probability of a head on an individual flip of the coin. Almost all interesting probability models for data, such as regression models, have multiple parameters. The Bayesian inference framework extends straightforwardly to such multiple-parameter models: Collect the parameters of the model in the vector $\boldsymbol{\phi} = (\phi_1, \phi_2, \ldots, \phi_p)$ and the observed data in $\mathbf{D}$, often a data matrix.[7] Then the likelihood function is $L(\boldsymbol{\phi}|\mathbf{D})$, the multivariate prior density is $p(\boldsymbol{\phi})$, and the multivariate posterior density for the parameters is, as in the single-parameter case, the normalized product of the likelihood and the prior density:

$$p(\boldsymbol{\phi}|\mathbf{D}) = \frac{p(\boldsymbol{\phi})L(\boldsymbol{\phi}|\mathbf{D})}{\int_{\boldsymbol{\phi}'} p(\boldsymbol{\phi}')L(\boldsymbol{\phi}'|\mathbf{D})d\boldsymbol{\phi}'}$$

[7] If you're unfamiliar with vector and matrix notation, just think of $\boldsymbol{\phi}$ as the list of parameters and $\mathbf{D}$ as a data table with rows representing cases and columns representing variables.

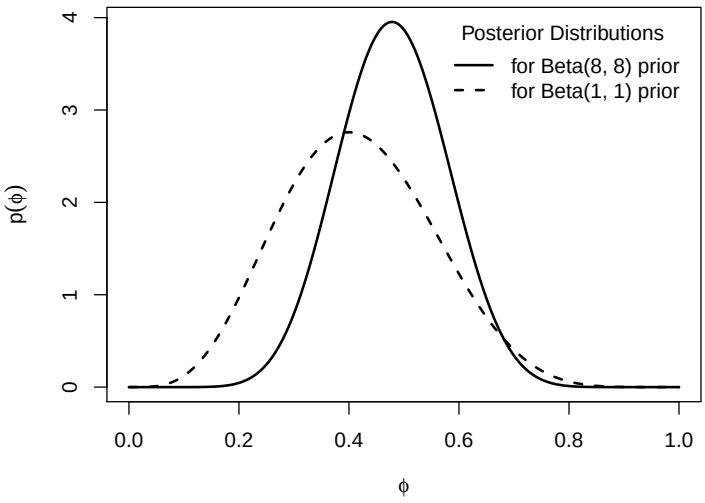

Figure 3: Posterior distributions for the coin-flipping experiment with 4 heads and 6 tails, corresponding to the noninformative Beta(1, 1) and informative Beta(8, 8) priors.

Inference then typically focuses on the *marginal posterior density* of each of the individual parameters ϕ_j in $\boldsymbol{\phi}$, for which, for example, we can formulate a point estimate, such as the posterior mean, and a credible interval.

2 Fitting Bayesian Models with Stan and rstan

Until fairly recently, practical Bayesian analysis was limited to applications with conjugate priors, for which evaluating the integral in the denominator of the posterior distribution is tractable. *Markov-Chain Monte Carlo* methods (abbreviated *MCMC*), a suite of random-sampling simulation methods, allow us to draw samples from the posterior distribution even when its analytic evaluation is impractical. MCMC methods were originally formulated in the 1950s to deal with similarly difficult probability calculations in statistical physics. Their application to statistics more generally, together with convenient software, has greatly expanded the application of Bayesian methods to data analysis.

In this section, we introduce the Stan software for Bayesian estimation, together with the **rstan** package, which provides a convenient interface to Stan from R. Stan implements a species of MCMC methods called *Hamiltonian Monte Carlo*, which provides advantages of robustness and efficiency relative to older MCMC methods such as the Metropolis-Hastings algorithm and the Gibbs sampler.[8]

Stan is a programming language for specifying and estimating probability models, a language with its own syntax and semantics, distinct from the syntax and semantics of R. Stan syntax is similar in certain respects to R and in other respects to C++. Indeed, a Stan program representing a probability model is translated into a corresponding C++ program, which is subsequently compiled into a directly executable program in machine code. The resulting executable program is typically

[8]See the complementary readings at the end of this appendix for sources that explain how MCMC sampling methods work.

much more efficient than an equivalent program written in interpreted R code. That's a good thing because MCMC methods are highly computationally intensive.

2.1 A Simple Example: The Coin-Flipping Experiment

We begin with a simple example, the Bernoulli-Beta model for the coin-flipping experiment that we introduced in the preceding section. Because the example uses a conjugate prior, we can easily derive the posterior distribution (as indeed we did) and we consequently don't have to use MCMC sampling methods. Let's see, however, whether we can get the right answer from Stan.

We begin by loading the **rstan** package. As the startup message for **rstan** indicates, the package is capable of taking advantage of parallel processing on a computer with multiple cores and can save compiled Stan programs to avoid the overhead of recompiling them. We take both these hints:

```
library(rstan)
```

```
Loading required package: ggplot2
```

```
Loading required package: StanHeaders
```

```
rstan (Version 2.17.3, GitRev: 2e1f913d3ca3)
```

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

```
(ncores <- parallel::detectCores())
```

```
[1] 4
```

```
options(mc.cores=ncores)
rstan_options(auto_write=TRUE)
```

We can define a Stan model either by typing the model into an R character string or by placing the code for the model into a file of file type .stan. The latter is a generally the more convenient choice for serious work, and the RStudio source-code editor recognizes and provides syntax highlighting for .stan files. For our simple preliminary example and more generally in this appendix, however, we write the model into a character string:

```
bern.model <- "
data {
    int<lower=0> n; // number of observations
    int<lower=0, upper=1> y[n]; // observations, 0=failure; 1=success
    int<lower=1> a; // shape parameter 1 for Beta prior
    int<lower=1> b; // shape parameter 2 for Beta prior
}
parameters {
    real<lower=0, upper=1> phi;  // per observation probability of success
}
model {
    phi ~ beta(a, b); // Beta(a, b) prior for phi
    y ~ bernoulli(phi); // likelihood
}
"
```

This example illustrates several characteristics of Stan programs:

- The program consists of several named *blocks* of code, here `data`, `parameters`, and `model`. Statements in each code block are enclosed in braces, `{ }`, and each statement ends with a semicolon, `;`.
- End-of-line comments are preceded by two slashes, `//`; everything to the right of the double-slashes is ignored by the Stan compiler.
- Unlike in R, variables in Stan must be declared before they are used. The declaration defines the kind of data that the variable contains, for example, integers (`int`) or floating-point numbers (`real`), and, if applicable, the dimensions of the variable are given in square brackets `[ ]`.
- The rules for naming variables in Stan are somewhat different from the rules in R. In particular, periods (`.`) aren't allowed in variable names and underscores (`_`) are conventionally used instead as a separator.
- If the variable is bounded, then the `lower` bound, `upper` bound, or both are specified in angle brackets `< >`.
- Stan includes a wide variety of probability functions. Probability distributions are specified using tilde (`~`) notation, *`variable ~ distribution(parameter1, parameter2, ...)`*

The model `bern.model` therefore has the following interpretation:

- The data input to be supplied to the model includes the non-negative integer number of observations `n`; a vector `y` of `n` integers, each either `0` or `1`, representing respectively "failure" and "success" (tails and heads for the coin-flipping experiment); and the two shape parameters `a` and `b` for the Beta prior, each an integer greater than or equal to 1.
- There is one parameter in the model, the probability `phi` of success (i.e., obtaining a head) on a single Bernoulli trial, which is a real number between zero and 1.
- The parameter `phi` follows a Beta prior with shape parameters `a` and `b`.
- The likelihood function for the observed data `y` is from the Bernouli distribution with probability of success `phi`.

Probability functions in Stan are vectorized (as they are in R), and so it's not necessary to define the likelihood for each component of `y`. It's not wrong, however, to do so, and we could replace the statement

```
y ~ bernoulli(phi);
```

in our program with a `for` loop, the syntax for which is very similar to R:

```
for (i in 1:n){
    y[i] ~ bernoulli(phi); // likelihood
}
```

Vectorized code in Stan generally runs more efficiently, and so we prefer the first formulation.

We proceed to collect the input for the model in an R list:

```
coin.exper <- list(n=10, y=c(1, 1, 0, 1, 0, 0, 0, 0, 1, 0), a=1, b=1) # flat prior
```

and generate a seed to use for Stan's random-number generator to make our results reproducible (as we did in the *R Companion* when we used simulation methods like the bootstrap):

```
sample(1e6, 1)
```

```
277360
```

Next, we call the `stan()` function in the **rstan** package to compile and fit the model, using the `system.time()` function to time the computation:[9]

```
system.time(
    bern.1 <- stan(
        model_code=bern.model,
        model_name="Bernoulli model, flat prior",
        seed=277360,
        data=coin.exper,
        iter=10000,
        chains=4
    )
)
```

```
   user  system elapsed
   0.97    0.37   45.86
```

The computation takes less than minute on our computer.[10]

The `stan()` function has a number of arguments (see `help("stan")`), several of which we use here:

`model_code` is a character variable containing the definition of the model; if the model were in a .stan file, we'd use the `file` argument instead.

`model_name` is an optional character string to label output.

`seed` is an optional seed for Stan's random-number generator.

`data` is an R list containing elements corresponding to the variables defined in the Stan program `data` block.

`iter` is the number of iterations to perform in each chain (see immediately below). Hamiltonian Monte-Carlo is an iterative method (as are MCMC methods generally) that samples repeated random draws from the posterior distribution. We specify 10,000 draws; the default is 2,000. Also by default, the first half of the sampled values are treated as a "warm-up" and are discarded. MCMC draws are generally not *independent* observations from the posterior distribution, and so the effective sample size after the warm-up period is less than the equivalent of `iter`/2 independent obervations. An advantage of Hamiltonian Monte Carlo over other MCMC methods is that it tends to produce samples with less-correlated observations.

`chains` is the number of MCMC chains to be run. The 4 chains produce samples that are independent of one-another. By comparing the chains, it's possible to learn whether or not the MCMC process probably converged to the target distribution (see below). The default number of chains is 4, and so we could have omitted this argument. It's possible to run the chains in parallel on a computer with multiple cores.

[9] The `rstan()` function normally reports its progress on the R console; we suppress the console output here.

[10] Most of this time is devoted to compiling the Stan program. Once compiled, the program takes about 5 seconds to run.

Printing the object returned by `stan()` summarizes the results; we call `print()` explicitly to get more than two places to the right of the decimal point:[11]

```
print(bern.1, digits=4)
```

```
Inference for Stan model: Bernoulli model, flat prior.
4 chains, each with iter=10000; warmup=5000; thin=1;
post-warmup draws per chain=5000, total post-warmup draws=20000.

        mean se_mean     sd     2.5%     25%     50%     75%   97.5% n_eff   Rhat
phi   0.4164  0.0016 0.1381   0.1634  0.3152  0.4121  0.5132  0.6971  7015 1.0007
lp__ -8.6836  0.0082 0.7491 -10.8838 -8.8409 -8.3979 -8.2078 -8.1509  8430 1.0003

Samples were drawn using NUTS(diag_e) at Tue Sep 18 17:14:08 2018.
For each parameter, n_eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor on split chains (at
convergence, Rhat=1).
```

- The results are reported for two quantities, `phi` and `lp__`. The former is the parameter of interest; the latter is the MCMC approximation to the log-posterior density, up to an additive constant, and isn't generally of direct interest.
- The column labelled `mean` contains the estimated mean of the posterior distribution of `phi` and the mean (adjusted) log-posterior density. The mean of the posterior distribution provides an estimate of ϕ, $\bar{\phi} = 0.4164$. As we know, the true posterior distribution is Beta(5, 7). The mean of a Beta distribution is $\alpha/(\alpha + \beta)$, which in this case is $5/(5 + 7) = 0.4167$.
- The column labelled `se_mean` is the *simulation* standard error of the quantity in the `mean` column; the simulation variability will decrease if we increase the number of iterations. In contrast, the column labelled `sd` is the *estimation* standard error of the `mean`. In this case, the simulation standard error for the posterior mean of ϕ is 0.0016, and thus it's not surprising that the MCMC mean and the mean of the true Beta(5, 7) posterior differ slightly. The estimation standard error of $\bar{\phi}$ is quite wide, 0.1381, as one would expect in a sample of only $n = 10$ cases coupled with a flat prior.
- The subsequent several columns give various percentiles of the posterior distribution; thus a 95% credible interval for ϕ is (0.1634, 0.6971).
- The column marked `n_eff` is the effective sample size, smaller than the $4 \times 5000 = 20,000$ values sampled after warm-up.
- The column labelled `Rhat` is a convergence diagnostic; if the marginal posterior distribution for a parameter has converged, then `Rhat` should be no greater than about 1.1.

Let's compare Stan's approximation to the posterior distribution of ϕ to the theoretically correct Beta(5, 7) posterior; to do so we call the `densityPlot()` function in the **car** package, producing Figure 4:

```
library("car")
```

```
Loading required package: carData
```

[11] The annotation `thin=1` in the output indicates that all sampled values after the burn-in period are retained; specifying the argument `thin=2` to `stan()`, for example, would discard every other sampled value, producing less-correlated observations from the posterior distribution at the cost of sampling fewer retained values. `NUTS(diag_e)`, also shown in the `stan()` output, indicates the kind of Hamiltonian Monte Carlo sampler that was employed.

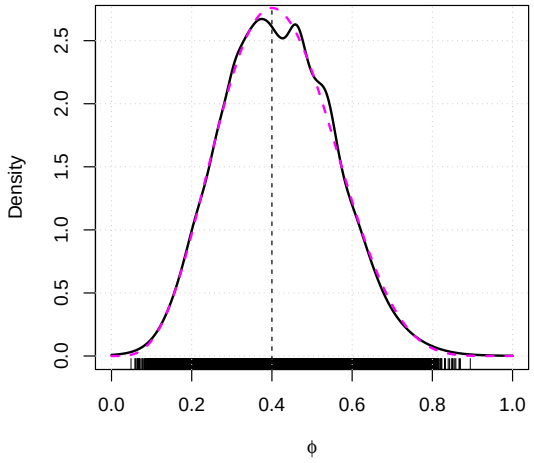

Figure 4: Comparison of the approximate posterior density produced by Stan (the solid black line) to the exact Beta(5, 7) posterior density (the broken magenta line), for the coin-flipping experiment with a flat prior.

```
densityPlot(extract(bern.1, "phi")$phi, from=0, to=1, normalize=TRUE,
            xlab=expression(phi))
lines(phi, dbeta(phi, 5, 7), col="magenta", lwd=2, lty=2)
abline(v=0.4, lty=2)
```

We use the `extract()` function from the **rstan** package to retrieve the 20,000 sampled values of ϕ; the function returns a list with one element, `$phi`. The simulated posterior density produced by Stan is very similar to the true Beta(5, 7) posterior density, albeit with a few small bumps.

Finally, it is prudent to plot the "traces" of the sampled values of ϕ over the 10,000 iterations for each of the 4 chains. If the chains have converged, then the traces should level out after the burn-in period and the chains should "mix"—that is, largely overlap one-another; we use the `traceplot()` function in **rstan** to draw the graph (in Figure 5):

```
traceplot(bern.1)
```

The results are unproblematic.

We invite the reader to redo this example with the informative Beta(8, 8) prior, simply changing the values of `a` and `b` in the `data` list passed to `stan()`.

2.2 The Normal Linear Regression Model

We're familiar from Chapter 4 of the *R Companion* with the normal linear regression model, which may be written

$$y_i|x_{1i}, \ldots, \beta_k x_{ki} \sim N(\beta_0 + \beta_1 x_{1i} + \cdots + \beta_k x_{ki}, \sigma^2)$$

where y_i is the response for the ith of n cases; the x_{ji}s are regressors for the ith case; the β_js (regression coefficients) and σ^2 (conditional variance of the response) are parameters to be estimated from the data; and $y_i, y_{i'}$ are independentfor $i \neq i'$. We're accustomed to estimating this model by least-squares regression. In this section, we show you how to use Stan to get Bayesian estimates for

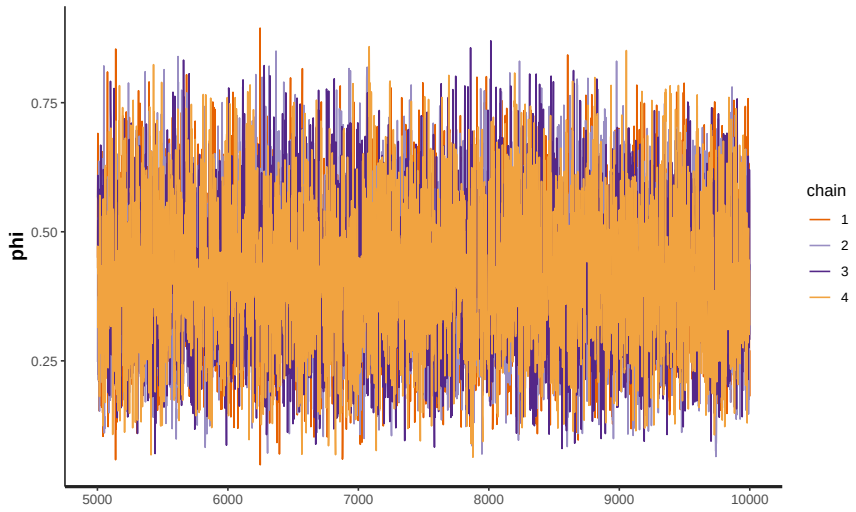

Figure 5: Post burn-in trace plot for the 4 chains used in the coin-flipping example with flat prior.

the parameters of the normal linear model, using Duncan's occupational prestige data (introduced in Section 1.5 of the *R Companion*) for an illustration. The data are in the `Duncan` data frame in the **carData** package, which has already been loaded in the current R session.

Here's a Stan model for Duncan's regression of `prestige` on `income` and `education`:

```
duncan.model.1 <- "
    data {
        int<lower=0> n;         // number of cases
        int<lower=0> k;         // number of regressors (less intercept)
        vector[n] prestige;     // response
        vector[n] income;       // predictor
        vector[n] education;    // predictor
    }
    parameters {
        vector[k + 1] beta;     // regression coefficients
        real<lower=0> sigma;    // error standard deviation
    }
    transformed parameters {
        vector[n] mu = beta[1] + beta[2]*income + beta[3]*education;
                    // conditional expectation of the response
    }
    model {
        prestige ~ normal(mu, sigma); // likelihood
    }
"
```

This model incorporates a few notable features:

- We code the number of cases `n` and number of predictors `k` as "data" to make it easier to generalize the model.
- We define the response variable, the predictors, and the regression parameters as Stan `vectors`. Because Stan vectors are indexed from 1, not zero, the intercept in the model is named

`beta[1]`.[12]

- The error standard deviation `sigma` is a real number bounded below by zero.
- We introduce a new Stan program block, `transformed parameters`, to define the vector of expectations `mu` of the response in terms of the regression coefficients and predictors. This isn't strictly necessary here, but it allows us to show how to define dependencies among the parameters of a model, and simplifies the expression for the distribution of the response.
- We declare the response `prestige` to be normally distributed with conditional mean `mu` and standard deviation (*not* variance!) `sigma`.
- We don't specify prior distributions for the regression coefficients and error standard deviation, and so Stan will use flat priors for these parameters. Because the regression coefficients are unbounded and the error standard deviation is unbounded above, flat priors on these parameters enclose an infinite area rather than an area of 1, and hence are not proper density functions. In many circumstances, including the current model, *improper priors* nevertheless lead to proper posterior distributions, and hence are considered by some to constitute a reasonable expression of prior ignorance about the values of parameters. We consider proper priors for the parameters of Duncan's regression model below.

Using the `stan()` function to fit the model is straightforward. As usual, we start by assembling a data list and we select a known seed for Stan's random-number generator:[13]

```
data.duncan <- with(Duncan, list(n=nrow(Duncan), k=2,
                    prestige=prestige, income=income, education=education))
```

```
sample(1e6, 1)
```

```
693089
```

```
system.time(fit.duncan.1 <- stan(model_code=duncan.model.1,
                      model_name="Duncan regression, flat priors",
                      seed=693089, data=data.duncan, iter=10000))
```

```
   user  system elapsed
   0.95    0.43   45.52
```

Let's compare the estimates of the regression parameters from Stan with those obtained by least-squares regression (which, with the exception of the error standard deviation, are maximum-likelihood estimates for the parameters of the normal linear model):[14]

```
print(fit.duncan.1, pars=c("beta", "sigma"), digits=4)
```

```
Inference for Stan model: Duncan regression, flat priors.
4 chains, each with iter=10000; warmup=5000; thin=1;
post-warmup draws per chain=5000, total post-warmup draws=20000.

          mean se_mean     sd     2.5%     25%     50%     75%   97.5% n_eff   Rhat
beta[1] -6.0613  0.0393 4.3867 -14.8444 -8.9543 -6.0377 -3.1539  2.4602 12485 1.0001
beta[2]  0.5991  0.0012 0.1234   0.3551  0.5164  0.6000  0.6821  0.8372 10471 1.0003
```

[12] Stan vectors are column vectors; it's also possible to define row vectors and matrices; see the Stan manual.

[13] As in our previous example, almost all of the time required for this command was used to compile the model.

[14] The maximum-likelihood estimate of σ^2, and hence of σ, is obtained by dividing the residual sum of squares by n rather than by residual degrees of freedom.

```
beta[3]  0.5457  0.0010 0.1004   0.3502  0.4786  0.5451  0.6132  0.7429 10347 1.0001
sigma   13.7799  0.0142 1.5811  11.1058 12.6747 13.6174 14.7293 17.3244 12398 0.9999

Samples were drawn using NUTS(diag_e) at Tue Sep 18 17:14:55 2018.
For each parameter, n_eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor on split chains (at
convergence, Rhat=1).
```

```
brief(lm(prestige ~ income + education, data=Duncan))
```

```
            (Intercept) income education
Estimate          -6.06  0.599    0.5458
Std. Error         4.27  0.120    0.0983

 Residual SD = 13.4 on 42 df, R-squared = 0.828
```

In printing the Bayesian estimates we specify the parameters `beta` and `sigma` to avoid displaying estimates of the $n = 45$ elements of `mu`, which are fitted values under the model. It's apparent that the flat-prior Bayesian and least-squares estimates and standard errors are very similar. Trace plots of the sampled parameter values (not shown) reveal no convergence issues.

Although it's not the case here, flat improper priors can be problematic, for example, producing convergence problems in Stan. An alternative is to use *diffuse*, but proper, *weakly informative priors*. Doing so, however, requires some thought about the scale and meaning of the parameters. For our current example, we start by rescaling the predictors `income` and `education` to zero means, so that the intercept of the regression model estimates the population mean of the response, `prestige`.

All of the variables in Duncan's regression are percentages, and so it doesn't make sense to have an intercept in the rescaled model less than zero or greater than 100. Although we could constrain the intecept to this range, we can instead simply pick a prior distribution for the intercept that has very little probability below zero or above 100. We use normal priors for this example because of their familiarity, but there are many other reasonable choices for weakly informative priors. We know that 68% of values are within 1 standard deviation of the mean of a normally distributed variable, 95% within 2 standard deviations, and 99.7% within 3 standard deviations. The weakly informative prior $N(50, 20^2)$ for β_0 is centered at the middle of the `prestige` scale and virtually rules out values below zero or above 100.

Similarly, it seems very improbable that a 1% increase in `income` or `education` would produce as much as a 2 or 3% change in `prestige`. Specifying the weakly informative prior $N(0, 3^2)$ for the `income` and `education` slopes makes wildly large positive or negative coefficients very improbable but allows a wide range of values with substantial prior density. It seems prudent, and is common practice in specifying diffuse priors, to center priors for slope coefficients at zero (i.e., no effect for the corresponding predictor).

These considerations lead to the following Stan model:

```
duncan.model.2 <- "
    data {
        int<lower=0> n;       // number of cases
        int<lower=0> k;       // number of regressors (less intercept)
        vector[n] prestige;   // response
        vector[n] income;     // predictor
        vector[n] education;  // predictor
    }
    transformed data{
        vector[n] income_d = income - mean(income);
```

```
        vector[n] education_d = education - mean(education);
    }
    parameters {
        vector[k + 1] beta;   // regression coefficients
        real<lower=0> sigma;  // error standard deviation
    }
    transformed parameters {
        vector[n] mu = beta[1] + beta[2]*income_d + beta[3]*education_d;
                    // conditional expectation of the response
    }
    model {
        prestige ~ normal(mu, sigma); // likelihood
        // priors:
        beta[1] ~ normal(50, 20);
        beta[2] ~ normal(0, 3);
        beta[3] ~ normal(0, 3);
        sigma ~ normal(0, 10);
    }
"
```

We also specify a vague normal prior centered at zero for the error standard deviation `sigma`; because `sigma` is constrained to be non-negative, only the right half of the normal distribution is used, producing a "half-normal" prior. In addition to specifying prior distributions for the parameters, we introduce a `transformed data` block into the Stan program, defining mean-deviation variables `income_d` and `education_d` for the two predictors.

Fitting the model proceeds as before, and as before, because the diffuse priors contribute little to the posterior, we obtain results similar to OLS regression:

```
fit.duncan.2 <- stan(model_code=duncan.model.2,
                     model_name="Duncan regression, weakly informative priors",
                     seed=693089, data=data.duncan, iter=10000)
```

```
print(fit.duncan.2, pars=c("beta", "sigma"), digits=4)
```

```
Inference for Stan model: Duncan regression, weakly informative priors.
4 chains, each with iter=10000; warmup=5000; thin=1;
post-warmup draws per chain=5000, total post-warmup draws=20000.

           mean se_mean     sd    2.5%     25%     50%     75%   97.5% n_eff   Rhat
beta[1] 47.7444  0.0159 2.0364 43.7312 46.3915 47.7411 49.0845 51.7910 16340 1.0001
beta[2]  0.5989  0.0011 0.1219  0.3561  0.5178  0.5985  0.6789  0.8370 11874 1.0000
beta[3]  0.5454  0.0009 0.0998  0.3481  0.4796  0.5455  0.6103  0.7441 11569 1.0002
sigma   13.4771  0.0117 1.4614 10.9930 12.4487 13.3447 14.3638 16.7165 15584 0.9999

Samples were drawn using NUTS(diag_e) at Tue Sep 18 17:15:40 2018.
For each parameter, n_eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor on split chains (at
convergence, Rhat=1).
```

```
brief(lm(prestige ~ I(income - mean(income)) + I(education - mean(education)),
         data=Duncan))
```

```
           (Intercept) I(income - mean(income)) I(education - mean(education))
Estimate         47.69                   0.599                          0.5458
```

```
Std. Error        1.99                    0.120                          0.0983

 Residual SD = 13.4 on 42 df, R-squared = 0.828
```

2.2.1 A More General Stan Program for Linear Models*

Stan supports matrix computations, and so using the matrix form of the normal linear model, $\mathbf{y}|\mathbf{X} \sim N_n(\alpha + \mathbf{X}\boldsymbol{\beta}, \sigma^2\mathbf{I}_n)$, allows us to write a more general linear-model Stan program. We treat the intercept α separately from the other regression parameters. Using flat priors, for example:

```
linmod <- "
    data {
        int<lower=0> n; // number of cases
        int<lower=0> k; // number of regressors, less constant
        matrix[n, k] X; // model matrix, less constant
        vector[n] y;    // response vector
    }
    parameters {
        vector[k] beta;      // regression coefficients
        real alpha;          // regression intercept
        real<lower=0> sigma; // residual standard deviation
    }
    transformed parameters{
        vector[n] mu = alpha + X*beta;
    }
    model {
        y ~ normal(mu, sigma); // likelihood
    }
"
```

Unlike in R, where `*` is always the elementwise product and `%*%` is used for the matrix product, in Stan the product operator `*` is used more generally for a variety of products including the matrix product, here for the matrix product `X*beta`.

Applying this model to Duncan's regression produces the same results (within the bounds of similation error) as before:

```
sample(1e6, 1)
```

```
332486
```

```
X <- model.matrix(~ income + education - 1, data=Duncan) # suppress intercept
head(X)
```

```
           income education
accountant     62        86
pilot          72        76
architect      75        92
author         55        90
chemist        64        86
minister       21        84
```

```
data.duncan.3 <- list(n=nrow(X), k=ncol(X), X=X, y=Duncan$prestige)
```

```
fit.duncan.3 <- stan(model_code=linmod,
             model_name="Duncan regression, matrix formulation, flat priors",
             seed=332486, data=data.duncan.3, iter=10000)

print(fit.duncan.3, pars=c("alpha", "beta", "sigma"), digits=4)

Inference for Stan model: Duncan regression, matrix formulation, flat priors.
4 chains, each with iter=10000; warmup=5000; thin=1;
post-warmup draws per chain=5000, total post-warmup draws=20000.

           mean se_mean     sd     2.5%     25%     50%     75%   97.5% n_eff   Rhat
alpha   -6.0250  0.0379 4.4274 -14.7224 -8.9623 -6.0327 -3.0519  2.5734 13647 1.0000
beta[1]  0.5986  0.0012 0.1249   0.3545  0.5167  0.5984  0.6823  0.8452 10634 1.0001
beta[2]  0.5455  0.0010 0.1012   0.3444  0.4785  0.5455  0.6125  0.7439 10260 1.0000
sigma   13.7968  0.0132 1.5479  11.1864 12.7046 13.6622 14.7452 17.2198 13671 0.9999

Samples were drawn using NUTS(diag_e) at Tue Sep 18 17:16:24 2018.
For each parameter, n_eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor on split chains (at
convergence, Rhat=1).

print(fit.duncan.1, pars=c("beta", "sigma"), digits=4)

Inference for Stan model: Duncan regression, flat priors.
4 chains, each with iter=10000; warmup=5000; thin=1;
post-warmup draws per chain=5000, total post-warmup draws=20000.

           mean se_mean     sd     2.5%     25%     50%     75%   97.5% n_eff   Rhat
beta[1] -6.0613  0.0393 4.3867 -14.8444 -8.9543 -6.0377 -3.1539  2.4602 12485 1.0001
beta[2]  0.5991  0.0012 0.1234   0.3551  0.5164  0.6000  0.6821  0.8372 10471 1.0003
beta[3]  0.5457  0.0010 0.1004   0.3502  0.4786  0.5451  0.6132  0.7429 10347 1.0001
sigma   13.7799  0.0142 1.5811  11.1058 12.6747 13.6174 14.7293 17.3244 12398 0.9999

Samples were drawn using NUTS(diag_e) at Tue Sep 18 17:14:55 2018.
For each parameter, n_eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor on split chains (at
convergence, Rhat=1).
```

Using the approach described in Section 10.10 of the *R Companion*, it's a relatively short step between `linmod` and an R function that uses formula notation to fit a Stan model; we assume flat priors, and adapt code from the `lm()` function to process the standard `formula`, `data`, `subset`, `na.action`, and `contrasts` arguments. Our function, which we call `bayeslm()` is shown in Figure 6. There's not much new here: We put the Stan model in the global variable `blm`; we could have made this variable local to the `bayeslm()` function, but the model will only be compiled once anyway. The `bayeslm()` function passes the ellipses argument (`...`) to `stan()` to allow the user to supply other arguments to the latter.

The `bayeslm()` function doesn't treat the regression intercept specially and simply returns the object produced by `stan()`. There's certainly room for improvement here, for example by breaking out the intercept, allowing predictors to be put in mean-deviation form, providing for the specification of proper priors, and returning a more R-like model object, with `coef()`, `vcov()`, etc., methods. We invite the reader to pursue these and other possibilities.[15]

[15]There are several R packages that provide this kind of standard modeling interface to Stan and other programs

```
blm <- "
    data {
        int<lower=0> n; // number cases
        int<lower=0> p; // number of regressors
        matrix[n, p] X; // model matrix
        vector[n] y;    // response vector
    }
    parameters {
        vector[p] beta; // regression coefficients
        real<lower=0> sigma; // residual std. dev.
    }
    transformed parameters {
        vector[n] mu = X*beta; // expectation of y
    }
    model {
        y ~ normal(mu, sigma); // likelihood
    }
"

bayeslm <- function(formula, data, subset, na.action, contrasts=NULL, ...){
    if (!require(rstan)) stop ("rstan package not available")
    cl <- match.call()
    mf <- match.call(expand.dots = FALSE)
    m <- match(c("formula", "data", "subset", "na.action"),
               names(mf), 0L)
    mf <- mf[c(1L, m)]
    mf$drop.unused.levels <- TRUE
    mf[[1L]] <- quote(stats::model.frame)
    mf <- eval(mf, parent.frame())
    mt <- attr(mf, "terms")
    y <- model.response(mf, "numeric")
    X <- model.matrix(mt, mf, contrasts)
    n <- length(y)
    p <- ncol(X)
    Data <- list(n=n, X=X, y=y, p=p)
    stan(model_code=blm, model_name="Linear Model", data=Data, ...)
}
```

Figure 6: An R function that uses formula notation to specify and fit a Stan normal linear model with flat priors.

We apply `bayeslm()` to Duncan's regression, once more producing results that are similar to those obtained previously:

```
sample(1e6, 1)
```

```
300732
```

```
fit.duncan.4 <- bayeslm(prestige ~ income + education, data=Duncan,
                        seed=300732, iter=10000)
```

```
print(fit.duncan.4, pars=c("beta", "sigma"), digits=4)
```

```
Inference for Stan model: Linear Model.
4 chains, each with iter=10000; warmup=5000; thin=1;
post-warmup draws per chain=5000, total post-warmup draws=20000.

          mean se_mean     sd     2.5%     25%     50%     75%   97.5% n_eff   Rhat
beta[1] -6.0788  0.0390 4.3987 -14.8002 -8.9757 -6.0328 -3.1995  2.5250 12720 1.0000
beta[2]  0.6009  0.0012 0.1237   0.3540  0.5185  0.6016  0.6842  0.8428 10138 1.0003
beta[3]  0.5441  0.0010 0.1018   0.3449  0.4767  0.5437  0.6123  0.7446  9448 1.0005
sigma   13.7786  0.0145 1.5657  11.0990 12.6782 13.6339 14.7194 17.2615 11656 1.0004

Samples were drawn using NUTS(diag_e) at Tue Sep 18 17:17:08 2018.
For each parameter, n_eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor on split chains (at
convergence, Rhat=1).
```

```
print(fit.duncan.3, pars=c("alpha", "beta", "sigma"), digits=4)
```

```
Inference for Stan model: Duncan regression, matrix formulation, flat priors.
4 chains, each with iter=10000; warmup=5000; thin=1;
post-warmup draws per chain=5000, total post-warmup draws=20000.

          mean se_mean     sd     2.5%     25%     50%     75%   97.5% n_eff   Rhat
alpha   -6.0250  0.0379 4.4274 -14.7224 -8.9623 -6.0327 -3.0519  2.5734 13647 1.0000
beta[1]  0.5986  0.0012 0.1249   0.3545  0.5167  0.5984  0.6823  0.8452 10634 1.0001
beta[2]  0.5455  0.0010 0.1012   0.3444  0.4785  0.5455  0.6125  0.7439 10260 1.0000
sigma   13.7968  0.0132 1.5479  11.1864 12.7046 13.6622 14.7452 17.2198 13671 0.9999

Samples were drawn using NUTS(diag_e) at Tue Sep 18 17:16:24 2018.
For each parameter, n_eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor on split chains (at
convergence, Rhat=1).
```

for Bayesian computation; see, e.g., the **brms** package (Bürkner, 2017), the **rstanarm** package (Stan Development Team, 2016), and, more generally, the CRAN Bayesian Inference task view at `https://cran.r-project.org/web/views/Bayesian.html`. We illustrate the use of the **rstanarm** package in Section 2.3. It's our view, however, expressed more fully below, that the principal advantage to using general Bayesian software like Stan is the ability to customize statistical models to the data at hand. From this perspective, interposing an additional, albeit more familiar, software layer between the user and Stan is of dubious value.

2.3 Hierarchical Regression Models

The most compelling applications of Bayesian methods aren't typically to standard regression models such as the normal linear model or generalized linear models,[16] but to problem where we want to specify a customized probability model for the data. We illustrate in this section with a hierarchical model for longitudinal data on exercise and eating disorders described by Davis et al. (2005) and discussed in Chapter 7 of the *R Companion*.

The data for the example are in the data frame `Blackmore` in the **carData** package:

```
brief(Blackmore, rows=c(5, 5)) # first and last subjects
```

```
945 x 4 data.frame (935 rows omitted)
    subject   age exercise   group
        [f]   [n]      [n]     [f]
1       100  8.00     2.71 patient
2       100 10.00     1.94 patient
3       100 12.00     2.36 patient
4       100 14.00     1.54 patient
5       100 15.92     8.63 patient
. . .
768     286  8.00     1.10 control
769     286 10.00     1.10 control
770     286 12.00     0.35 control
771     286 14.00     0.40 control
772     286 17.00     0.29 control
```

```
nrow(Blackmore)
```

```
[1] 945
```

```
with(Blackmore, by(subject, group, function(x) length(unique(x))))
```

```
group: control
[1] 93
------------------------------------------------------------
group: patient
[1] 138
```

Recall that the data, for 138 girls hospitalized for eating orders and 93 similar-age control subjects, are in long form, with several observations at various ages for each girl. The variable `exercise` in the data set is estimated number of hours of exercise per week. Because `exercise` is positively skewed and has zero values, we transformed this variable using the Box-Cox-with-negatives family:

```
Blackmore$tran.exercise <- bcnPower(Blackmore$exercise, lambda=0.25, gamma=0.1)
```

In Section 7.2.5 of the *R Companion*, we fit several mixed-effects models to the `Blackmore` data, treating transformed `exercise` as the response, and the between-subject factor `group` and the within-subject covariate `age` as predictors, including in our models the interaction between `group` and `age`. We entertained the possibility that different subjects had different trajectories, that is subject-specific intercepts and slopes, treated as random effects, and that the observation-level

[16] There are exceptions. For example, incorporating well-founded prior beliefs, based, say, on existing research, might provide much more useful estimates for sparse data sets than do traditional methods. Similarly, specifying a vague prior for logistic regression coefficients can avoid infinite estimates in the face of separable data.

errors were autocorrelated. Our estimates of the fixed effects didn't depend strongly on the random-effects structure that we specified, and here we'll focus on the model with random intercepts and slopes but conditionally independent observation-level errors:[17]

```
library(lme4)
```

```
Loading required package: Matrix
```

```
blackmore.mod.lmer <- lmer(tran.exercise ~ I(age - 8)*group +
    (I(age - 8) | subject), data=Blackmore, REML=FALSE)
S(blackmore.mod.lmer)
```

```
Linear mixed model fit by ML
Call: lmer(formula = tran.exercise ~ I(age - 8) * group + (I(age - 8) | subject),
          data = Blackmore, REML = FALSE)

Estimates of Fixed Effects:
                       Estimate Std. Error z value Pr(>|z|)
(Intercept)            -0.17194    0.12653  -1.359  0.17418
I(age - 8)              0.06025    0.02322   2.595  0.00945
grouppatient           -0.22578    0.16321  -1.383  0.16656
I(age - 8):grouppatient  0.20114    0.02921   6.887 5.71e-12

Estimates of Random Effects (Covariance Components):
 Groups   Name        Std.Dev. Corr
 subject  (Intercept) 0.9918
          I(age - 8)  0.1314   -0.06
 Residual             0.8787

Number of obs: 945, groups:  subject, 231

  logLik       df      AIC      BIC
-1492.23        8  3000.46  3039.27
```

As in the text, we subtract 8 from `age` so that the regression intercept estimates transformed exercise at `age` 8, the start of the study. In contrast to the text, and although it makes virtually no difference, we fit the model here by ML rather than by REML for greater similarity to the Bayesian models we entertain below.

Although we eventually want to deal explicitly with the zero-`exercise` values in the `Blackmore` data, let's start by specifying a Stan model similar to the model fit by `lmer()`:

```
blackmore.model.1 <- "
    data {
        int<lower=0> n;          // total number of observations
        int<lower=0> J;          // number of subjects
        int subject[n];          // subject index
        vector[n] patient;       // patient dummy regressor
        vector[n] age;           // within-subject predictor age - 8 yrs
        vector[n] tran_exercise; // response variable, transformed exercise
    }
```

[17] We omit autocorrelated errors to simplify the specification of Bayesian models for the `Blackmore` data, not because Stan can't accommodate autocorrelated errors—it can. As mentioned, in the text we obtained similar results for a mixed model with random intercepts and slopes.

```
    parameters {
        matrix[J, 2] B;         // subject-specific intercepts and slopes
                                //     (deviations)
        vector[4] beta;         // group-average intercepts and slopes
        real<lower=0> sigma;    // error standard deviation
        corr_matrix[2] Rho;     // correlation matrix of subject-specific
                                //     coefficients
        vector<lower=0>[2] psi; // standard deviations of subject-specific
                                //     coefficients
    }
    transformed parameters {
        matrix[2, 2] Psi = quad_form_diag(Rho, psi);
                            // covariance matrix of subject-specific
                            //     coefficients
        real rho = Rho[1, 2]; // correlation between subject-specific
                            //     intercepts and slopes
        vector[n] mu = beta[1] + B[subject, 1] + beta[2]*patient + beta[3]*age
                        + beta[4] * patient .* age + B[subject, 2] .* age;
                            // conditional expectation of response
    }
    model{
        tran_exercise ~ normal(mu, sigma); // conditional distribution of response
        for (j in 1:J){
            B[j, ] ~ multi_normal(rep_vector(0, 2), Psi);
                                    // distribution of subject-specific coefficents
        }
        // priors:
        beta  ~ normal(0, 10);
        sigma ~ normal(0, 10);
        psi ~ normal(0, 10);
        Rho ~ lkj_corr(2);
    }
"
```

This hierarchical model is considerably more complex than the Stan regression models we entertained previously in this appendix and consequently requires some additional explanation:

- The definitions in the `data` block of the Stan program are largely straightforward; we define the number of observations `n` (rows in the `Blackmore` data set) and the number of subjects `J`, along with the variables that appear in the model. The variable `patient` is a 0/1 dummy regressor, `age` is age - 8 years, `tran_exercise` is transformed exercise, and `subject` is coded as successive integers, 1 through `J`.
- The parameters of the model include the `J` $\times$ 2 matrix `B` of subject-specific intercepts and slopes, expressed as deviations from group-average intercepts and slopes. In a traditional mixed-effects model, we'd call these coefficients "random effects," but in the Bayesian framework all unknown values are random variables, and so we avoid the term here.
- There is also the vector `beta` of 4 coefficients to code the group-average intercepts and slopes; in a traditional mixed model, these would be "fixed effects."
- The parameter `sigma` represents the standard deviation of the observation-level errors.

- We also need parameters for the standard deviations and correlation of the subject-level intercept and slope coefficients. These are respectively in the vector `psi`, with 2 elements, and in the 2-by-2 correlation matrix `Rho`. Although `Rho` contains only one unknown parameter—the correlation between the intercepts and the slopes—we formulate the model in this way to insure that the covariance matrix of the subject-level coefficients is consistent (*i.e., positive-definite). We separate the coefficient standard deviations from their correlation, rather than specifying their covariance matrix directly, to make it simpler to formulate priors for these parameters (see below). `Rho` and `psi` are termed *hyperparameters* because they pertain to the distribution of the rows of `B`, which are themselves unknown values ("parameters").

- In the `transformed parameters` block, we define the covariance matrix `Psi` of the subject-specific regression coefficients in terms of the coefficient standard deviations and their correlation matrix. If you're unfamiliar with matrix arithmetic, disregard the following:* The expression `quad_form_diag(Rho, psi)` computes the matrix product diag(`psi`) `Rho` diag(`psi`).

- Using the group-average and subject-specific regression coefficients along with the data for the predictors, we compute the conditional expectation `mu` of the response for each observation. The computation is fully vectorized, which requires in places the Stan elementwise product operator `.*` to multiply corresponding elements of two vectors—in R we'd just use `*` for this operation.

- The `model` block defines the conditional normal distribution of the response, and specifies that the subject-specific intercept and slope for each subject are bivariately normally distributed with zero expectations and covariance matrix `Psi` defined previously. This specification requires a loop over the `J` subjects.

- The group-level regression coefficients `beta`, the observation-level error standard deviation `sigma`, and the subject-specific coefficient standard deviations `psi` are all given weakly informative normal priors; for the standard deviations, which are constrained to be non-negative, these are half-normal priors.

- Finally, the prior for the correlation matrix `Rho` of the subject-specific coefficients is the LKJ distribution with shape parameter 2. *The LKJ distribution insures that `Rho` is positive-definite with 1s down the diagonal, as is appropriate for a correlation matrix. The shape parameter is similar to the shape parameter for a symmetric Beta(α, α) distribution: LKJ(α) is centered at zero correlation and is more spread out as α gets smaller. LKJ(1) is essentially a flat prior for the correlation, while LKJ(2) is weakly informative. The LKJ distribution generalizes to larger correlation matrices, beyond the bivariate case. See the Stan manual for details.

We proceed to assemble the data for the Stan hierarchical model:

```
data.blackmore <- with(Blackmore, list(n = nrow(Blackmore),
    J = nlevels(subject),
    subject = as.numeric(subject),
    patient = as.numeric(group == "patient"),
    age = age - 8,
    tran_exercise = tran.exercise))
```

and then fit the model using a known randomly selected seed (whose selection is not shown) for Stan's random-number generator, timing the computation (which takes quite a long time, more than 10 minutes):

```
system.time(fit.blackmore.1 <- stan(model_code=blackmore.model.1, chains=4,
                model_name= "Hierarchical model for Blackmore data, weak priors",
```

```
                seed=702478, data=data.blackmore, iter=10000,
                control=list(max_treedepth=20)))

   user  system elapsed
   4.20    1.43  636.86

print(fit.blackmore.1, pars=c("beta", "sigma", "psi", "rho"), digits=4)

Inference for Stan model: Hierarchical model for Blackmore data, weak priors.
4 chains, each with iter=10000; warmup=5000; thin=1;
post-warmup draws per chain=5000, total post-warmup draws=20000.

           mean se_mean     sd    2.5%     25%     50%     75%  97.5% n_eff   Rhat
beta[1] -0.1720  0.0014 0.1270 -0.4231 -0.2564 -0.1728 -0.0869 0.0786  8378 1.0005
beta[2] -0.2257  0.0019 0.1646 -0.5489 -0.3354 -0.2257 -0.1130 0.0931  7905 1.0007
beta[3]  0.0606  0.0002 0.0234  0.0150  0.0447  0.0606  0.0765 0.1061 13157 1.0001
beta[4]  0.2008  0.0003 0.0296  0.1428  0.1807  0.2007  0.2208 0.2590 13110 1.0002
sigma    0.8840  0.0003 0.0284  0.8304  0.8646  0.8834  0.9026 0.9415  6675 1.0008
psi[1]   1.0000  0.0009 0.0714  0.8648  0.9509  0.9985  1.0473 1.1434  6292 1.0003
psi[2]   0.1314  0.0004 0.0174  0.0967  0.1198  0.1316  0.1432 0.1653  2202 1.0021
rho     -0.0415  0.0027 0.1354 -0.2818 -0.1366 -0.0491  0.0432 0.2481  2599 1.0015

Samples were drawn using NUTS(diag_e) at Mon Oct 01 13:52:34 2018.
For each parameter, n_eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor on split chains (at
convergence, Rhat=1).
```

These results based on our weak priors are very similar to the ML estimates produced by `lmer()` (see page 21).

As we mentioned (in footnote 15 on page 17), the **rstanarm** package provides an R-familiar front end to Stan for a variety of standard statistical models, including linear models, generalized linear models, and hierarchical models. Here, for example, is how we'd use `stan_lmer()` from the **rstanarm** package to obtain Bayesian estimates for the hierarchical model we're entertaining for the `Blackmore` data, not bothering to set the seed for the random number generator and taking all defaults:

```
library("rstanarm")

system.time(blackmore.mod.2.stanlmer <- stan_lmer(tran.exercise ~ I(age - 8)*group +
    (I(age - 8) | subject), data=Blackmore))

   user  system elapsed
   3.69    0.33  134.75

print(blackmore.mod.2.stanlmer, digits=4)

stan_lmer
 family:       gaussian [identity]
 formula:      tran.exercise ~ I(age - 8) * group + (I(age - 8) | subject)
 observations: 945
------
                      Median  MAD_SD
(Intercept)           -0.1826  0.1239
```

```
I(age - 8)               0.0612  0.0236
grouppatient            -0.2171  0.1630
I(age - 8):grouppatient  0.2000  0.0290
sigma                    0.8824  0.0283

Error terms:
 Groups   Name        Std.Dev. Corr
 subject  (Intercept) 0.99340
          I(age - 8)  0.13398  -0.051
 Residual             0.88258
Num. levels: subject 231

Sample avg. posterior predictive distribution of y:
         Median MAD_SD
mean_PPD 0.3177 0.0407

------
For info on the priors used see help('prior_summary.stanreg').
```

By default, `stan_lmer()` employs weakly informative priors (better thought-out than those we specified above) and so we get similar results to those that we obtained previously. The `stan_lmer()` function is substantially faster than our Stan program, but we used 4 chains of 10,000 iterations each, while `stan_lmer()` used the default 4 chains of 2,000 iterations each, accounting for the difference.

Having invoked Stan essentially to duplicate our previous mixed-model analysis of the `Blackmore` data, let's now use it to better advantage to formulate a customized model that makes explicit provision for the zero exercise values in the data. We define a so-called "hurdle" model, similar in spirit to the zero-inflated Poisson (ZIP) regression model described in Section 10.6.1 of the *R Companion*.

The model consists of two parts: A logit model for the probability of nonzero exercise, and a linear model for transformed exercise given that exercise is nonzero.[18] Both parts of the model have group-level and subject-specific regression coefficients. We encountered minor numerical difficulties in fitting the model with both subject-specific intercepts and slopes, and so simpified the subject-specific part of the model to include just intercepts. The Stan model is then as follows:

```
blackmore.model.2 <- "
    data {
        int<lower=0> n;                    // total number of observations
        int<lower=0> J;                    // number of subjects
        int subject[n];                    // subject index
        vector[n] patient;                 // patient dummy regressor
        vector[n] age;                     // within-subject predictor
        vector[n] tran_exercise;           // response variable

        int<lower=0, upper=1> nonzero[n];  // exercise > 0?
        int n_nonzero;                     // number of nonzero observations
        int<lower=1, upper=n>
            index_nonzero[n_nonzero];      // indices of nonzero observations

        // for effect display:
```

[18] Because our model now treats zero exercise separately, we don't really need the Box-Cox-with-negatives transformation family—we could use a traditional Box-Cox transformation—but stick with it for comparability to our previous results.

```
    int<lower=0> effect_n;              // number of values for effect display
    vector[effect_n] patient_effect;    // for effect display
    vector[effect_n] age_effect;        // for effect display
}
parameters {
    vector[J] b;         // subject-specific intercepts
    vector[4] beta;      // group-level coefficients
    real<lower=0> sigma; // error standard deviation
    real<lower=0> psi_b; // standard deviation of intercepts
    vector[J] z;         // subject-specific intercepts for logit part of model
    vector[4] zeta;      // group-level coefficients for logit part of model
    real<lower=0> psi_z; // standard deviation of subject-specific
                         //     logit intercepts
}
transformed parameters {
    // conditional expectation for linear part of model
    vector[n] mu = beta[1] + b[subject] + beta[2] * patient
            + beta[3] * age + beta[4] * patient .* age;
    // linear predictor for for logit part of model
    vector[n] eta = zeta[1]+ z[subject] + zeta[2] * patient
            + zeta[3] * age + zeta[4] * patient .* age;
}
model{
    nonzero ~ bernoulli_logit(eta);         // logit part of model
    tran_exercise[index_nonzero]
        ~ normal(mu[index_nonzero], sigma); // linear part of model
    b ~ normal(0, psi_b);       // distribution of intercepts for linear model
    z ~ normal(0, psi_z);       // distribution of intercepts for logit model
    // priors:
    sigma ~ normal(0, 10);
    beta  ~ normal(0, 10);
    zeta  ~ normal(0, 10);
    psi_b ~ normal(0, 10);
    psi_z ~ normal(0, 10);
}
generated quantities{
    vector[effect_n] eta_effect = zeta[1] + zeta[2] * patient_effect
            + zeta[3] * age_effect + zeta[4] * patient_effect .* age_effect;
    vector[effect_n] mu_effect = beta[1] + beta[2] * patient_effect
            + beta[3] * age_effect + beta[4] * patient_effect .* age_effect;
    mu_effect =  mu_effect .* (1. ./ (1. + exp( - eta_effect)));
}
"
```

The model has several notable new features:

- We add the 0/1 variable `nonzero` to the `data` block to serve as the response in the logit part of the model. We also add `n_nonzero`, the count of the number of zonzero observations, and `index_nonzero`, the indices of the nonzero observations. These variables could have been computed in a `transformed data` block in the Stan program, but we find it easier to do these kinds of computations in R and to pass the results to Stan as "data."

- Also in the `data` block, we define quantitities that we'll use to create an effect plot for the fitted model, giving combinations of ages (`age_effect`) and groups (`patient_effect`), and the number of such combinations (`effect_n`). Again, we find it convenient to compute these quantitites in R and to pass them to Stan as data.
- The parameters for the linear part of the model are similar to those in `blackmore.model.1`, except that we now only have subject-specific intercepts, in the vector `b`, with standard deviation `psi_b`.
- The parameters for the logit part of the model parallel those for the linear part, and include the group-level regression coefficients `zeta`, subject-specific intercepts `z`, and the standard deviation of the intercepts `psi_z`.
- In the `transformed parameters` block we compute both the conditional expectation `mu` of the response for the linear part of the model and the linear predictor `eta` for the logit part of the model.
- The `model` block uses the `bernoulli_logit()` function in Stan to define the likelihood for the logit part of the model. The normal linear part of the model is the same as before, except that it now applies only to observations where exercise is nonzero. The remainder of the model is straightforward, with normal distributions for the subject-specific intercepts in both parts of the model, and diffuse normal priors.
- Finally, we introduce a new `generated quantities` block into the program to calculate fitted transformed exercise for an effect display of the model. We first compute the fitted logit `eta_effect` of nonzero exercise and the fitted transformed exercise `mu_effect`, and then multiply the latter by the fitted probability of nonzero exercise computed from the former. In transforming the fitted logits into probabilities, we use the Stan `.*` and `./` operators for elementwise multiplication and division of vectors. The computation of these values could be performed, perhaps more easily, in R after fitting the model, but we want to demonstrate the use of the `generated quantities` Stan program block.

We need to add several new variables to the R data list to be passed to Stan, both for the hurdle model and to compute the effect display:

```
data.blackmore$nonzero <- as.numeric(Blackmore$exercise > 0)
data.blackmore$n_nonzero <- sum(data.blackmore$nonzero)
data.blackmore$index_nonzero <- which(as.logical(data.blackmore$nonzero))

data.blackmore$effect_n <- 12
(data.blackmore$patient_effect <- rep(0:1, each=6))

 [1] 0 0 0 0 0 0 1 1 1 1 1 1

(data.blackmore$age_effect <- rep(seq(8, 18, by=2), 2) - 8)

 [1]  0  2  4  6  8 10  0  2  4  6  8 10
```

We then fit and summarize the model as before, using a new precomputed seed for Stan's random-number generator:

```
fit.blackmore.2 <- stan(model_code=blackmore.model.2, chains=4,
                        model_name= "Hurdle model for Blackmore data, weak priors",
                        seed=620314, data=data.blackmore, iter=10000)
```

```
print(fit.blackmore.2,
      pars=c("beta", "zeta", "sigma", "psi_b", "psi_z"),
      digits=4)

Inference for Stan model: Hurdle model for Blackmore data, weak priors.
4 chains, each with iter=10000; warmup=5000; thin=1;
post-warmup draws per chain=5000, total post-warmup draws=20000.

          mean se_mean     sd    2.5%     25%     50%     75%  97.5% n_eff   Rhat
beta[1] -0.0227  0.0015 0.1182 -0.2528 -0.1032 -0.0234  0.0567 0.2100  6084 1.0002
beta[2] -0.1226  0.0020 0.1529 -0.4221 -0.2254 -0.1225 -0.0194 0.1753  5841 1.0003
beta[3]  0.0812  0.0001 0.0179  0.0463  0.0692  0.0811  0.0932 0.1163 16852 1.0000
beta[4]  0.1743  0.0002 0.0224  0.1309  0.1592  0.1744  0.1895 0.2180 15451 1.0001
zeta[1]  3.3239  0.0078 0.4445  2.5148  3.0137  3.3046  3.6173 4.2479  3223 1.0023
zeta[2] -0.8036  0.0045 0.4788 -1.7687 -1.1150 -0.7989 -0.4781 0.1143 11352 1.0001
zeta[3] -0.0793  0.0006 0.0708 -0.2175 -0.1267 -0.0794 -0.0327 0.0607 14422 0.9999
zeta[4]  0.2066  0.0008 0.0894  0.0314  0.1464  0.2049  0.2664 0.3830 13996 1.0000
sigma    0.8197  0.0002 0.0238  0.7752  0.8033  0.8193  0.8354 0.8679 20000 0.9999
psi_b    0.8873  0.0004 0.0528  0.7887  0.8503  0.8861  0.9215 0.9957 20000 1.0000
psi_z    1.7687  0.0079 0.2801  1.2705  1.5708  1.7526  1.9453 2.3625  1242 1.0064

Samples were drawn using NUTS(diag_e) at Wed Sep 19 19:50:42 2018.
For each parameter, n_eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor on split chains (at
convergence, Rhat=1).
```

The transformation of `exercise` and the two equations of the hurdle model make it difficult to interpret the results directly from the regression coefficients and so we will draw an effect plot for the fitted model. We begin by using the `extract()` function from the **rstan** package to obtain the 10,000 simulated values of the effects, and then compute means and quantiles from these values:

```
mu <- extract(fit.blackmore.2, pars=c("mu_effect"))$mu_effect
mu.fit <- colMeans(mu)
mu.025 <- apply(mu, 2, quantile, probs=0.025)
mu.975 <- apply(mu, 2, quantile, probs=0.975)
age.effect <- rep(seq(8, 18, by=2), 2)
patient.effect <- factor(rep(c("control", "patient"), each=6))
Effect <- data.frame(exercise=bcnPowerInverse(mu.fit, lambda=0.25, gamma=0.1),
                     exercise.025=bcnPowerInverse(mu.025, lambda=0.25, gamma=0.1),
                     exercise.975=bcnPowerInverse(mu.975, lambda=0.25, gamma=0.1),
                     group=patient.effect, age=age.effect)
Effect

    exercise exercise.025 exercise.975   group age
1  0.9757751    0.7750219     1.216435 control   8
2  1.1383867    0.9333547     1.378213 control  10
3  1.3176798    1.0893709     1.581059 control  12
4  1.5130135    1.2370141     1.835463 control  14
5  1.7227085    1.3723402     2.143509 control  16
6  1.9437319    1.4991721     2.501834 control  18
7  0.8697040    0.7209473     1.039100 patient   8
8  1.3883191    1.1936427     1.607588 patient  10
```

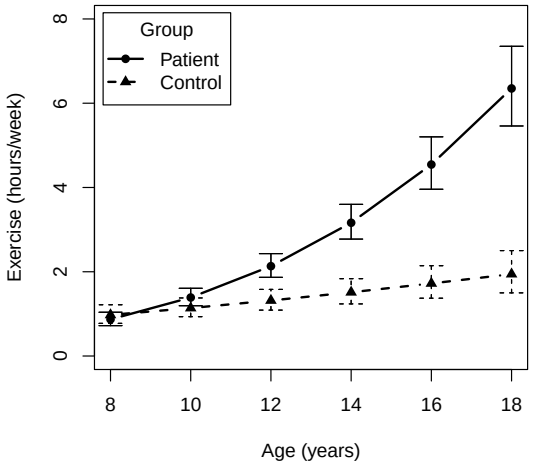

Figure 7: Effect plot for the Bayesian hierarchical hurdle model fit to the `Blackmore` data.

```
9  2.1325411    1.8673624     2.430411 patient  12
10 3.1625469    2.7748462     3.599344 patient  14
11 4.5445067    3.9578084     5.201936 patient  16
12 6.3504911    5.4597067     7.350821 patient  18
```

We use the `bcnPowerInverse()` function to undo the transformation of the response, restoring `exercise` to the hours/week scale.

The final step is to draw the effect display, shown in Figure 7:

```
plot(c(8, 18), c(0, 8), type="n", xlab="Age (years)",
     ylab="Exercise (hours/week)")
lines(exercise ~ age, data=Effect, subset=group=="control",
      type="b", pch=17, lty=2, lwd=2)
lines(exercise ~ age, data=Effect, subset=group=="patient",
      type="b", pch=16, lwd=2)
with(Effect[1:6, ], arrows(age.effect, exercise.025, age.effect,
                           exercise.975, angle=90, lty=2, code=3, length=0.1))
with(Effect[7:12, ], arrows(age.effect, exercise.025, age.effect,
                           exercise.975, angle=90, code=3, length=0.1))
legend("topleft", title="Group", legend=c("Patient", "Control"), lwd=2, lty=1:2,
       pch=c(16, 17), inset=0.02)
```

This graph is similar to the effect plot in Figure 7.10 (on page 369) of the text, computed from the linear mixed-effects model that we fit to transformed exercise, although the fitted values of exercise are now a bit larger.

3 Complementary Reading and References

The recent flowering of Bayesian methods is reflected in a vast literature on the subject. Here are some key sources:

- Gelman and Hill (2007) present Bayesian statistical methods from first principles in the context of regression modeling, including multilevel models. Computing is through R and BUGS, an older Bayesian modeling program whose name is an acronmym for Bayesian inference Using Gibbs Sampling) and which was once state-of-the-art. Andrew Gelman is one of the prime movers behind the development of Stan, and an updated two-volume version of this text that uses Stan has been in the works for some time.
- McElreath (2016) is a relatively gentle introduction to Bayesian methods, including regression and multilevel models, and not assuming a strong statistical background. As the subtitle of the text implies, computing is through R and Stan, largely using McElreath's somewhat idiosyncractic **rethinking** R package, which is freely available but not on CRAN.
- Gelman et al. (2013) is a wide-ranging, and more demanding, presentation of modern Bayesian methods. The book, now in its third edition, includes an appendix on using Stan and R.
- The Stan manual (Stan Development Team, 2017), available at `https://github.com/stan-dev/stan/releases/download/v2.17.0/stan-reference-2.17.0.pdf`, is a valuable and extensive source of information; also see a recent article in the *Journal of Statistical Software*, Carpenter et al. (2017), for a briefer introduction to Stan.

Appendix: Acquiring and Installing C++ Compilers for Windows and macOS Systems

In order to use Stan it's necessary to have a compatible C++ compiler installed on your computer. Linux/Unix systems typically already have C++ installed. On Windows and macOS systems, you'll have to install it, but in both cases installation is simple.

Windows

Go to your favorite CRAN mirror and click on Download R for Windows > Rtools. Alternatively, go directly to `https://cloud.r-project.org/bin/windows/Rtools/`. In the Rtools Downloads table, select a version of the Rtools installer that's compatible with your version of R, typically the most recent version of the installer if you have the current version of R.

After the Rtools*xy*.exe installer is downloaded (where *xy* represents the version of the installer), run it in the normal manner, for example, by double-clicking on the downloaded file in your web browser or in the File Explorer. You may take all of the default selections during the installation, but do make sure to allow the installer to add Rtools to your system path.

macOS

On macOS you'll have to install Apple's Xcode developer tools, which are available for free through the App Store, and can be installed from there in the usual manner.

References

Bürkner, P.-C. (2017). brms: An R package for Bayesian multilevel models using Stan. *Journal of Statistical Software*, 80(1):1–28.

Carpenter, B., Gelman, A., Hoffman, M., Lee, D., Goodrich, B., Betancourt, M., Brubaker, M., Guo, J., Li, P., and Riddell, A. (2017). Stan: A probabilistic programming language. *Journal of Statistical Software, Articles*, 76(1):1–32.

Davis, C., Blackmore, E., Katzman, D. K., and Fox, J. (2005). Female adolescents with anorexia nervosa and their parents: A case-control study of exercise attitudes and behaviours. *Psychological Medicine*, 35:377–386.

Fox, J. (2009). *A Mathematical Primer for Social Statistics*. Sage, Thousand Oaks CA.

Fox, J. and Weisberg, S. (2019). *An R Companion to Applied Regression*. Sage, Thousand Oaks, CA, third edition.

Gelman, A., Carlin, J. B., Stern, H. S., Dunson, D. B., Vehtari, A., and Rubin, D. S. (2013). *Bayesian Data Analysis*. Chapman and Hall/CRC Press, Boca Raton FL.

Gelman, A. and Hill, J. (2007). *Data Analysis Using Regression and Multilevel/Hierarchical Models*. Cambridge University Press, Cambridge UK.

McElreath, R. (2016). *Statistical Rethinking: A Bayesian Course with Examples in R and Stan*. Chapman and Hall/CRC Press, Boca Raton FL.

Stan Development Team (2016). rstanarm: Bayesian applied regression modeling via Stan. R package version 2.13.1.

Stan Development Team (2017). *Stan Modeling Language: User's Guide and Reference Manual, Stan Version 2.17.0*.

Stan Development Team (2018). rstan: the R interface to Stan. R package version 2.17.3.