CHAPTER 14

Stan

Contents

Fools lob proposals on random trajectories,
Finding requital just once in a century.
True love homes in on a heart that is radiant,
Guided by stars from Sir Hamilton's gradient.[1]

Stan is the name of a software package that creates representative samples of parameter values from a posterior distribution for complex hierarchical models, analogous to JAGS. Take a look at Figure 8.1, p. 194, which shows the relation of R to JAGS and Stan. Just as we can specify models for JAGS and communicate with JAGS from R via rjags, we can specify models for Stan and communicate with Stan from R via RStan.

According to the Stan reference manual, Stan is named after Stanislaw Ulam (1909–1984), who was a pioneer of Monte Carlo methods. (Stan is not named after the slang term referring to an overenthusiastic or psychotic fanatic, formed by a combination of the words "stalker" and "fan.") The name of the software package has also been unpacked as the acronym, Sampling Through Adaptive Neighborhoods (Gelman et al., 2013, p. 307), but it is usually written as Stan not STAN.

Stan uses a different method than JAGS for generating Monte Carlo steps. The method is called *Hamiltonian Monte Carlo* (HMC). HMC can be more effective than the

[1] This chapter is about an MCMC sampling scheme that creates proposal distributions that are pulled toward the mode(s) of the posterior distribution instead of being symmetrical around the current position. The proposals use trajectories based on the gradient of the posterior, using a mathematical scheme named after the physicist Sir William Hamilton.

Doing Bayesian Data Analysis, 2nd Edition
http://dx.doi.org/10.1016/B978-0-12-405888-0.00014-3

various samplers in JAGS and BUGS, especially for large complex models. Moreover, Stan operates with compiled C++ and allows greater programming flexibility, which again is especially useful for unusual or complex models. For large data sets or complex models, Stan can provide solutions when JAGS (or BUGS) takes too long or fails. However, Stan is not universally faster or better (at this stage in its development). For some of the applications in this book, JAGS works as fast or faster, and there are some models that cannot (yet) be directly expressed in Stan.

Stan involves a little extra programming overhead than JAGS, so Stan is a little harder to learn from scratch than JAGS. But once you know JAGS, it is fairly easy to learn the additional details of Stan. Stan also has extensive documentation and a number of programming abilities not available in JAGS. Because Stan is undergoing rapid development at the time of this writing, the goal of this book is not to present a complete library of programs for Stan. Instead, this chapter presents the ideas, a complete example, and guidelines for composing programs in Stan. Later chapters include a variety of additional Stan programs. By the time you learn Stan, some of its details may have changed.

14.1. HMC SAMPLING

Stan generates random representative samples from a posterior distribution by using a variation of the Metropolis algorithm called HMC. To understand it, we briefly review the Metropolis algorithm, which was explained in Section 7.3, p. 156. In the Metropolis algorithm, we take a random walk through parameter space, favoring parameter values that have relatively high posterior probability. To take the next step in the walk, there is a proposed jump from the current position, with the jump sampled randomly from a proposal distribution. The proposed jump is accepted or rejected probabilistically, according to the relative densities of the posterior at the proposed position and the current position. If the posterior density is higher at the proposed position than at the current position, the jump is definitely accepted. If the posterior density is lower at the proposed position than the current position, the jump is accepted only with probability equal to the ratio of the posterior densities.

The key feature I want to emphasize here is the shape of the proposal distribution: In the vanilla Metropolis algorithm, the proposal distribution is symmetrically centered on the current position. In multidimensional parameter spaces, the proposal distribution could be a multivariate Gaussian distribution, with its variances and covariances tuned for the particular application. But the multivariate Gaussian is always centered on the current position and is always the same shape regardless of where the walk roams in parameter space. This fixedness can lead to inefficiencies. For example, in the tails of the posterior distribution, the proposals will just as often go away from a posterior mode as go toward it, and therefore proposals will often be rejected. As another example, if the posterior distribution curves through parameter space, a fixed-shape proposal distribution that is

well tuned for one part of the posterior may be poorly tuned for another part of the posterior.

HMC instead uses a proposal distribution that changes depending on the current position. HMC figures out the direction in which the posterior distribution increases, called its gradient, and warps the proposal distribution toward the gradient. Consider Figure 14.1. The top panel shows a simple posterior distribution on a single parameter called theta. The current position in the Markov chain is indicated by a large dot on the abscissa. The two columns of Figure 14.1 show two different current positions. From either current position, a jump is proposed. In vanilla Metropolis, the proposal distribution would be a Gaussian centered on the current position, such that jumps above or below the current position would be equally likely to be proposed. But HMC generates proposals quite differently.

HMC generates a proposal by analogy to rolling a marble on the posterior distribution turned upside down. The second row of Figure 14.1 illustrates the upside-down posterior distribution. Mathematically, the upside-down posterior is the negative logarithm of the posterior density, and it is called the "potential" function for reasons to be revealed shortly. Wherever the posterior is tall, the potential is low, and wherever the posterior is short, the potential is high. The large dot, that represents the current position, is resting on the potential function like a marble waiting to roll downhill. The proposed next position is generated by flicking the marble in a random direction and letting it roll around for a certain duration. In this simple one-parameter example, the direction of the initial flick is randomly to the right or to the left, and the magnitude of the flick is randomly sampled from a zero-mean Gaussian. The flick imparts a random initial *momentum* to the ball, as suggested by the annotation in the second row of Figure 14.1. When time is up, the ball's new position is the proposed position for the Metropolis jump. You can imagine that the marble will tend to be caught at positions that are downhill on the potential function relative to the starting position. In other words, the proposed position will tend to be in regions of higher posterior probability.

The terminology comes from analogy to the physics of objects in gravity. In theoretical physics, a moving object has *kinetic* energy that trades off with *potential* energy: A stationary marble at the top of a hill has lots of potential energy but no kinetic energy, but when it is rolling to the bottom of the hill the marble has exchanged some of its potential energy for kinetic energy. In idealized frictionless systems, the sum of potential and kinetic energy is constant, and the dynamics of the system conserve the total energy. Real rolling balls violate the ideal because they incur friction (and change their angular momentum as they roll, which complicates the dynamics). A better analogy to the ideal is a puck sliding (not rolling) on a virtually frictionless surface such as smooth ice or a cushion of pressurized air. But ice feels cold and cushions of air involve noisy machines, so let's just imagine a pleasantly perfect marble on a smooth hill that has no friction and no spin.

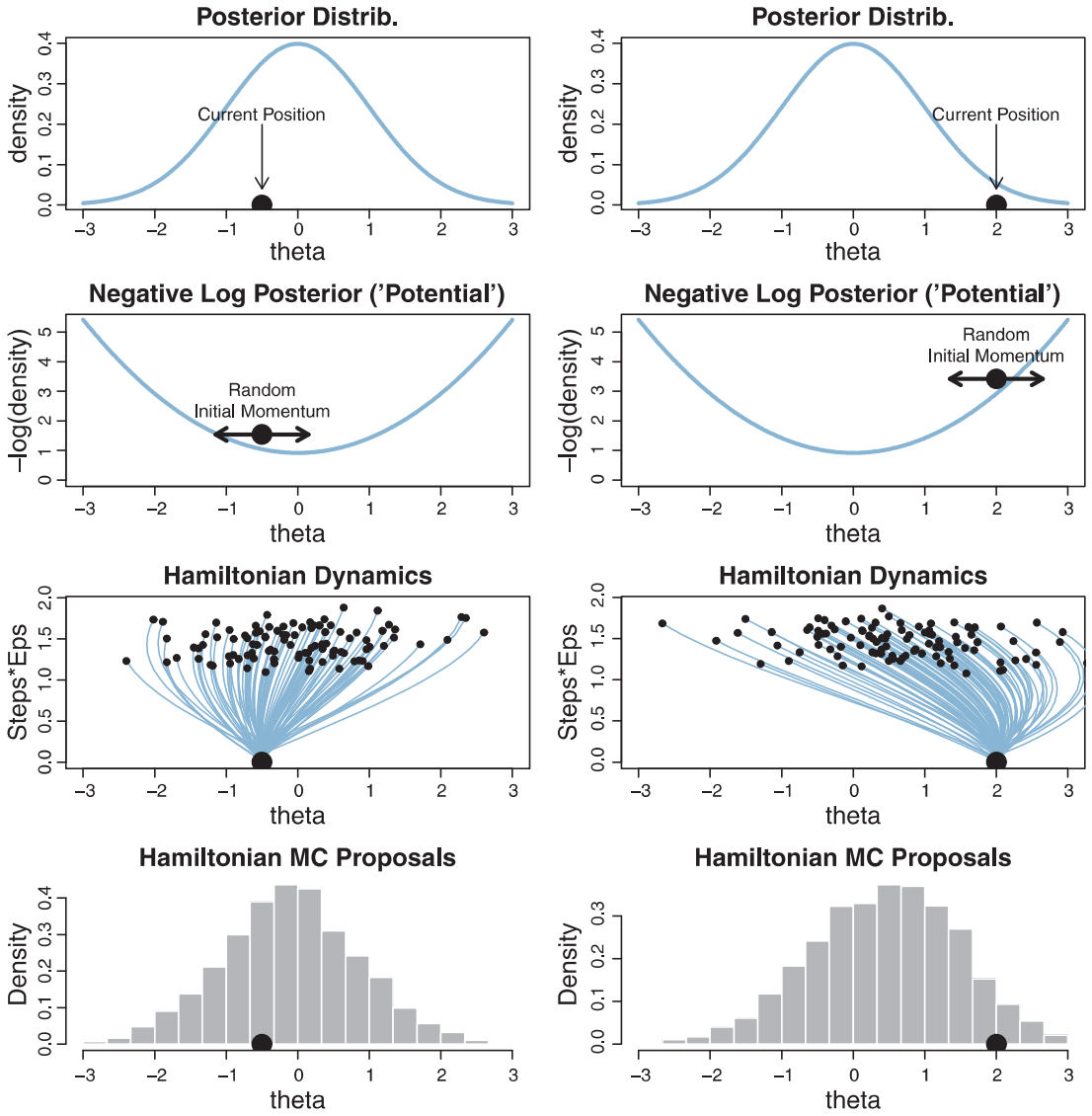

Figure 14.1 Examples of a Hamiltonian Monte Carlo proposal distributions. Two columns show two different current parameter values, marked by the large dots. First row shows posterior distribution. Second row shows the potential energy, with a random impulse given to the dot. Third row shows trajectories, which are the theta value (*x*-axis) as a function of time (*y*-axis marked Steps*Eps). Fourth row shows histograms of the proposals.

The third row of Figure 14.1 shows examples of many trajectories taken by balls that start on the current position and are given random impulses. The ordinate (*y*-axis) is mysteriously labeled as "Steps*Eps." Its exact meaning will be revealed below, but you can think of it simply as duration. Time increases as Steps*Eps increases. The trajectories in Figure 14.1 have random durations constrained within a fairly narrow

range of possibilities. The trajectories show the theta value as a function of time as the ball rolls after it receives the random initial impulse. The end point of each trajectory is marked with a small dot; this is the proposed position.

The bottom row of Figure 14.1 shows histograms of all the proposed positions. In particular, you can see that the proposal distribution is *not* centered on the current position. Instead, the proposal distribution is shifted toward the mode of the posterior distribution. Notice that the proposal distributions are quite different for the different current positions. In both cases, however, the proposals are shifted toward the mode of the posterior distribution. You can imagine that *for high-dimensional posterior distributions that have narrow diagonal valleys and even curved valleys, the dynamics of HMC will find proposed positions that are much more promising than a vanilla symmetric proposal distribution, and more promising than Gibbs sampling which can get stuck at diagonal walls (as was described at the very end of Section 7.4.4).*

Once the proposed jump is established, then the proposal is accepted or rejected according to the Metropolis decision rule as in Equation 7.1, p. 151, *except* that the terms involve not only the relative posterior density, but also the momentum at the current and proposed positions. The initial momentum applied at the current position is drawn randomly from a simple probability distribution such as a normal (Gaussian). Denote the momentum as ϕ. Then the Metropolis acceptance probability for HMC becomes

$$p_{\text{accept}} = \min\left(\frac{p(\theta_{\text{proposed}}|D)\, p(\phi_{\text{proposed}})}{p(\theta_{\text{current}}|D)\, p(\phi_{\text{current}})}, 1\right) \tag{14.1}$$

In an idealized continuous system, the sum of potential and kinetic energy [corresponding to $-\log(p(\theta|D))$ and $-\log(p(\phi))$] is constant, and therefore the ratio in Equation 14.1 would be 1, and the proposal would never be rejected. But in practical simulations, the continuous dynamics are discretized into small intervals of time, and the calculations are only approximate. Because of the discretization noise, the proposals will not always be accepted.

If the discrete steps of the trajectory are very small, then the approximation to the true continuous trajectory will be relatively good. But it will take many steps to go very far from the original position. Conversely, larger individual steps will make a poorer approximation to the continuous mathematics, but will take fewer steps to move far from the original position. Therefore, the proposal distribution can be "tuned" by adjusting the step size, called epsilon or "eps" for short, and by adjusting the number of steps. We think of the step size as the time it takes to make the step, therefore the total duration of the trajectory is the number of steps multiplied by the step size, or "Steps*Eps" as displayed on the y-axis of the trajectories in Figure 14.1. Practitioners of HMC typically strive for an acceptance rate of approximately 65% (Neal, 2011, p. 142). If the acceptance rate of a simulation is too low, then epsilon is reduced, and if the acceptance rate of

a simulation is too high, then epsilon is increased, with compensating changes in the number of steps to maintain the trajectory duration.

The step size controls the smoothness or jaggedness of the trajectory. The overall duration, Steps*Eps, controls how far the proposal ventures from the current position. This duration is important to tune, because we want the proposal to be closer to a mode, without overshooting, and without rolling all the way back to the starting point. Figure 14.2 shows a wide range of trajectories for the same starting positions as Figure 14.1. Notice that the long trajectories overshoot the mode and return to the current position. To prevent inefficiencies that would arise from letting the trajectories make a U-turn, Stan incorporates an algorithm that generalizes the notion of U-turn to high-dimensional parameter spaces and estimates when to stop the trajectories before they make a U-turn back toward the starting position. The algorithm is called the "no U-turn sampler" (NUTS; M. Hoffman & Gelman, 2014). By comparing the proposal distributions in Figures 14.1 and 14.2, you can intuit that the ones in Figure 14.1 will explore the posterior distribution more efficiently.

Other than step size and number of steps, there is a third tuning knob on the proposal distribution, namely, the standard deviation of the distribution from which the initial momentum is selected. Figure 14.3 shows examples of proposal distributions starting at the same current position, with the same trajectory duration, but with different standard deviations for the random initial momentum. You can see that when the standard deviation of the momentum distribution is wider, the proposal distribution is wider. As you can see by comparing Figures 14.1 and 14.3, the most efficient standard deviation is not too wide and not too narrow. The standard deviation of the momentum distribution is typically set by adaptive algorithms in Stan to match the standard deviation of the posterior.

Computing a proposal trajectory involves simulating the rolling of the marble down the potential hillside. In other words, we must be able to compute the gradient (i.e., derivative) of the posterior density, at any value of the parameter. This is done efficiently on high-dimensional parameter spaces only with explicit formulas for the gradient (as opposed to using numerical approximation by finite differentials). The formula could be derived by a human being, but for complex models with hundreds of parameters the formulas are derived algorithmically by symbolic-math syntactical engines. In simulating a proposal trajectory with discrete steps, the usual method is to first take a half step along the gradient to update the momentum, before alternating full steps along the gradients of potential and momentum, and finishing with another half step of momentum. This is called a "leapfrog" procedure because of the half steps.

The algorithms for computing gradients and for tuning proposal trajectories are sophisticated and complex. There are many technical hurdles, including a general-purpose system for analytically finding derivatives, dealing with limited-range parameters

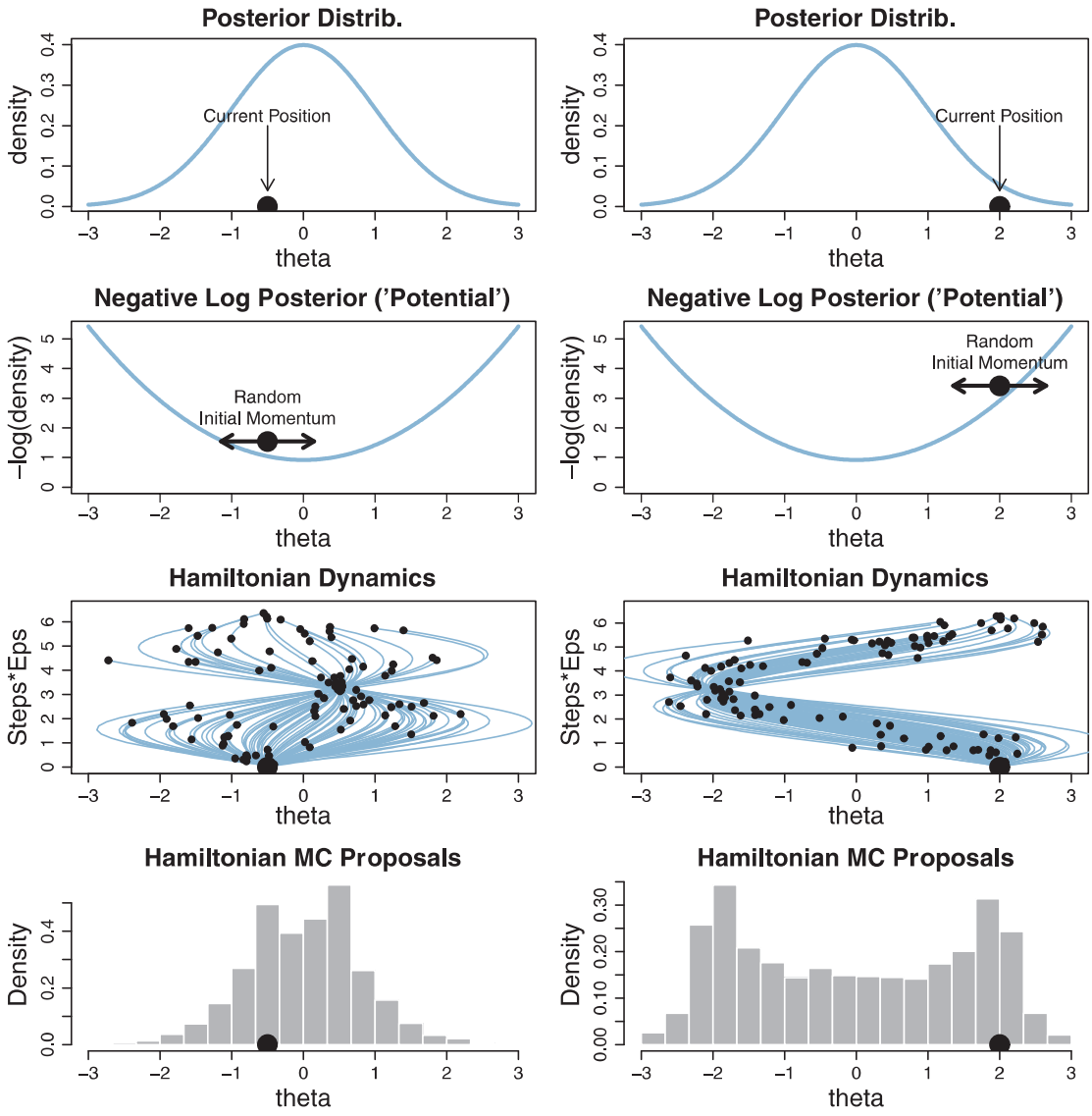

Figure 14.2 Examples of a Hamiltonian Monte Carlo proposal distributions for two different current parameter values, marked by the large dots, in the two columns. For this figure, a large range of random trajectory lengths (Steps*Eps) is sampled. Compare with Figure 14.1.

of different types, and various ways to tune the discretization of the Hamiltonian dynamics in high-dimensional parameter spaces. As Gelman et al. (2013, p. 307) suggest in this understatement, "Hamiltonian Monte Carlo takes a bit of effort to program and tune." Fortunately, the Stan system makes it relatively easy for the user.

Mathematical theories that accurately describe the dynamics of mechanical systems have been worked out by physicists. The formulation here, in terms of kinetic and

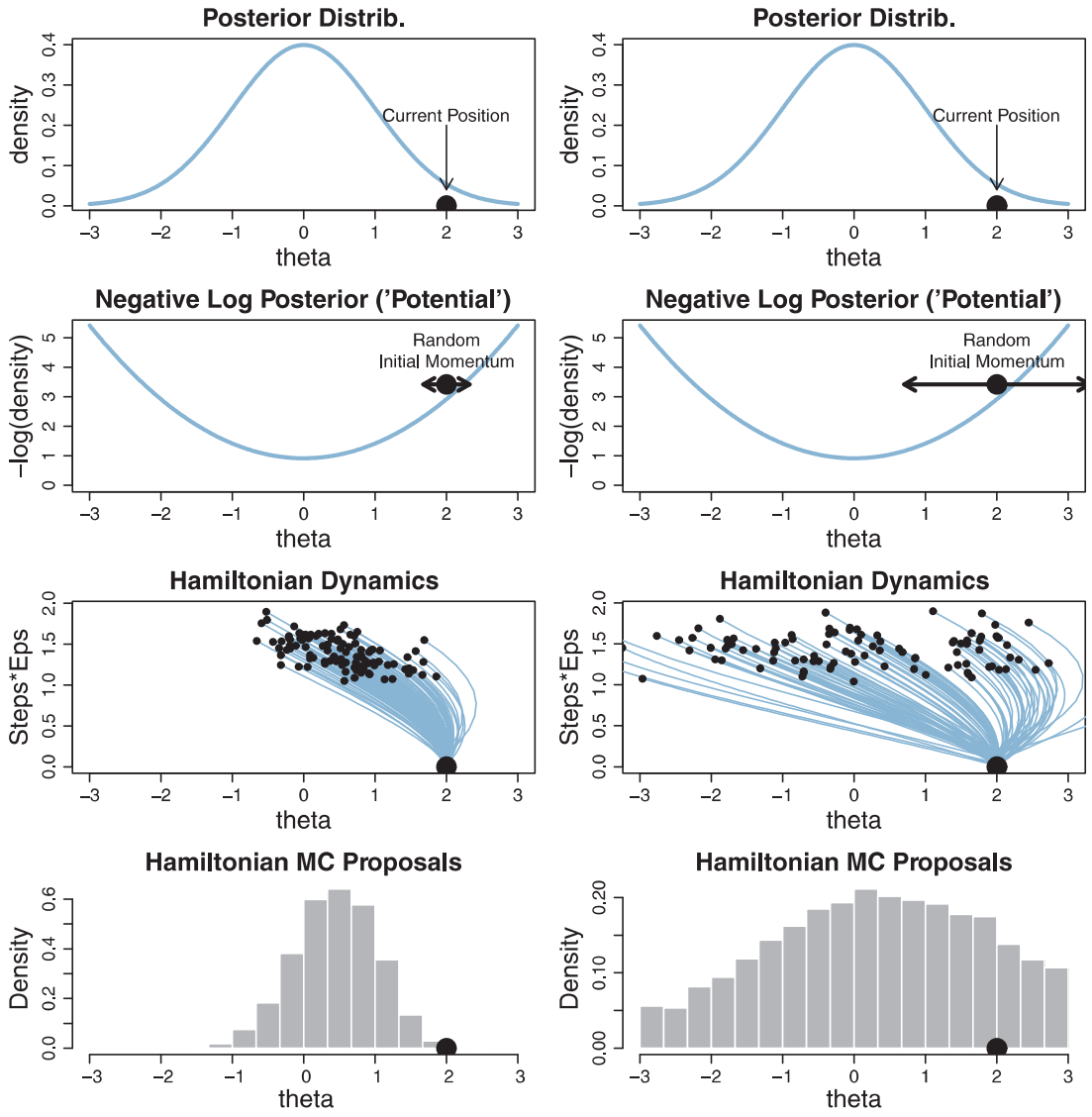

Figure 14.3 Examples of a Hamiltonian Monte Carlo proposal distributions for two different variances of the initial random momentum, indicated in the second row. Compare with Figure 14.1, which shows an intermediate variance of the initial random momentum.

potential energy, is named after William Rowan Hamilton (1805-1865). HMC was described in the physics literature by Duane, Kennedy, Pendleton, and Roweth (1987) (who called it "hybrid" Monte Carlo), and HMC was applied to statistical problems by Neal (1994). A brief mathematical overview of HMC is presented by MacKay (2003, chap. 30). A more thorough mathematical review of HMC is provided by Neal (2011). Details of how HMC is implemented in Stan can be found in the Stan reference manual and in the book by Gelman et al. (2013).

14.2. INSTALLING STAN

Go to the Stan home page at http://mc-stan.org/. (Web site addresses occasionally change. If the address stated here does not work, please search the web for "Stan language." Be sure the site is legitimate before downloading anything to your computer.) On that page there is a link to RStan. Click that link, and you will see extensive installation instructions. Be careful to follow every step in detail. Be sure that your versions of R and RStudio are up to date. I will not include specific details here because those details will change rapidly as Stan continues to be developed. Be sure to download the Stan modeling language user's guide and reference manual. When the RStan library is loaded in R, most of its functions have help pages. For example, if you want to learn more about the RStan `sampling` command, you can type `?sampling` at R's command line.

14.3. A COMPLETE EXAMPLE

The best way to explain programming in Stan with RStan is to show an example. We begin with the simple script named `Stan-BernBeta-Script.R`. The script implements the model of Figure 8.2, p. 196. The model estimates the bias of a single coin. The data are flips of a coin, $y_i \in \{0, 1\}$, described by a Bernoulli distribution, $y_i \sim \text{dbern}(\theta)$, with a beta prior, $\theta \sim \text{beta}(A, B)$. Like JAGS with rjags, the model for Stan with RStan is specified as a string in R. Unlike JAGS, however, the model specification begins with explicit declarations of which variables are data and which variables are parameters, in separately marked blocks before the model statement. Each variable declaration also states what numeric type the variable is, and any restrictions on its domain. For example, the declaration

```
int<lower=0> N ;
```

means that N is an integer value that has a lower bound of zero. Here is the complete model specification, enclosed as a string in R:

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

You will have noticed that every line within a block ends with an explicit end-of-command marker, which is a semi-colon. This is the syntax used in C++, which is the underlying language used by Stan. R also interprets a semi-colon as an end-of-command marker, which can be useful for typing several short commands on a single line. But whereas R recognizes either a carriage return or a semi-colon as the end of a command, Stan and C++ recognize only a semi-colon as the end of a command.

Another difference from JAGS is that comments in Stan are indicated by a double slash "`//`" instead of by a number sign "#." This difference again stems from the fact that Stan specifications are compiled into C++ code, and comments in C++ are indicated by a double slash.

In the model specification, above, the statements in the model block look analogous to the form used by JAGS and BUGS because that was a conscious design principle for Stan. But the Stan model specification is not identical to JAGS. For example, the probability densities are denoted as `beta` instead of `dbeta` and as `bernoulli` instead of `dbern`.

Another important difference is that Stan allows, in fact encourages, vectorization of operations. Thus, in Stan we can write a single line to indicate that every y_i value comes from the Bernoulli distribution:

```
y ~ bernoulli(theta) ;
```

But in JAGS we would have to write an explicit `for` loop:

```
for ( i in 1:N ) {
  y[i] ~ dbern(theta)
}
```

Stan does have `for` loops, but processing is faster when operations can be vectorized.

With the model specified as discussed above, the next step is to translate the model into C++ code and compile the C++ code into an executable *dynamic shared object* (DSO). The command in RStan for doing this is `stan_model`. Before it is called, however, the RStan library must be loaded into R:

```
library(rstan)
stanDso = stan_model( model_code=modelString )
```

If there were errors in the model, Stan would tell you about them here. This translation and compilation step can take a while, depending on how complex the model is. One of the key things that Stan is doing at this point is figuring out the gradient functions for the Hamiltonian dynamics. The resulting DSO is assigned, above, to the variable named `stanDso`.

Once the DSO is created, it can be used for generating a Monte Carlo sample from the posterior distribution. First we specify the data exactly as we did for JAGS, and then we generate the MC sample with the `sampling` command:

```
# Create some fictitious data:
N = 50 ; z = 10 ; y = c(rep(1,z),rep(0,N-z))
dataList = list( y = y , N = N )

stanFit = sampling( object=stanDso , data=dataList ,
                    chains=3 , iter=1000 , warmup=200 , thin=1 )
```

The arguments of the `sampling` command start with telling Stan what DSO to use. The next arguments should look familiar from rjags or runjags, except that Stan uses "`warmup`" instead of "`burnin`." In Stan, `iter` is the total number of steps per chain, including `warmup` steps in each chain. Thinning merely marks some steps as not to be used; thinning does not increase the number of steps taken. Thus, the total number of steps that Stan takes is `chains·iter`. Of those steps, the ones actually used as representative have a total count of `chains·(iter−warmup)/thin`. Therefore, if you know the desired total steps you want to keep, and you know the warm-up, chains, and thinning, then you can compute that the necessary `iter` equals the desired total multiplied by `thin/chains+warmup`.

We did not specify the initial values of the chains in the example above, instead letting Stan randomly initialize the chains by default. The chains can be initialized by the user with the argument `init`, analogous to JAGS. For more information, type "`?sampling`" at R's command line (after previously loading RStan with `library(rstan)`).

The `sampling` command returns more information than only the MC sample of representative parameter values. Also included is the DSO (again), along with information about the run details. There are various ways to examine the MC sample itself. RStan has methods for the standard R `plot` and `summary` commands, and RStan also has its own version of the `traceplot` command (for which there is a different version in the `coda` package used by JAGS). You can experiment with them easily from the command line in R. The RStan versions of `traceplot` and `plot` have an argument, `pars`, which takes a vector of strings that specify which parameters you want to plot. Here, we will convert that output of Stan into a `coda` object so that we can view the results using the same graphical format as we have been using for JAGS:

```
# Load rjags, coda, and DBDA2E functions:
source("DBDA2E-utilities.R")
# Convert stan format to coda format:
mcmcCoda = mcmc.list( lapply( 1:ncol(stanFit) ,
                              function(x) { mcmc(as.array(stanFit)[,x,]) } ) )
# Graph chain diagnostics using DBDA2E function:
diagMCMC( mcmcCoda , parName=c("theta") )
```

The resulting graph, not shown here, has the same format as Figure 8.3, p. 204.

14.3.1. Reusing the compiled model

Because model compilation can take a while in Stan, it is convenient to store the DSO of a successfully compiled model and use it repeatedly for different data sets. It is trivial to do this; just use the `sampling` command again with whatever data set is appropriate. This ability is especially useful for power analysis, which runs the analysis on many simulated data sets. Exercise 14.2 suggests how.

14.3.2. General structure of Stan model specification

The example presented above is very simple for the purpose of a first introduction to Stan. For more complex models, additional components of Stan model specification will be needed. The general structure of model specifications in Stan consist of six blocks, as suggested in the following outline:

```
data {
... declarations ...
}
transformed data {
... declarations ... statements ...
}
parameters {
... declarations ...
}
transformed parameters {
... declarations ... statements ...
}
model {
... declarations ... statements ...
}
generated quantities {
... declarations ... statements ...
}
```

Notice, for example, that after the `data` block there can be a `transformed data` block, in which statements can be placed that transform the data from the `data` statement into new values that can be used in the subsequently specified model. Analogously, parameters declared in the `parameters` block can be transformed to other parameters via statements in the `transformed parameters` block. At the end, if you want to monitor quantities generated from the model at each step, such as predictive values, you can create these in the `generated quantities` block.

The various blocks are optional (except the model block itself), but must be in the order shown above. As explained in detail in Section 14.4, the lines in a Stan model specification are processed in order. Therefore a variable must be declared

before it is used in a statement. In particular, that explains why the `transformed parameters` block must be placed after the `parameters` block. For an example, see Exercise 14.1.

14.3.3. Think log probability to think like Stan

If you think a bit about the algorithm used by Stan, you realize that most of Stan's effort is spent on computing the trajectory for Hamiltonian dynamics. From a current parameter position, the algorithm randomly generates an initial momentum and jittered step size and number of steps. Then the Hamiltonian dynamics are deterministically computed from the gradient of the potential function, that is, from the gradient of the (negative) logarithm of the posterior density. This repeats for many steps. At the end of the trajectory, the ratio of the posterior density at proposed and current positions (along with the momentum) is used to deterministically compute an acceptance probability. A random number from a uniform distribution is sampled if the acceptance probability is less than 1. Notice that the probability of acceptance in Equation 14.1 (p. 403) could be re-written by taking the logarithm of both sides, so that the acceptance probability also involves computing the logarithm of the posterior density. Thus, the essence of computation in Stan is dealing with the logarithm of the posterior probability density and its gradient; there is no direct random sampling of parameters from distributions.

If there is no random sampling of parameters from distributions, then what could a model specification like `y ~ normal(mu,sigma)` actually mean to Stan? It means to multiply the current posterior probability by the density of the normal distribution at the datum value `y`. Equivalently, it means to increment the current log-probability by the log-density of the normal at the datum. In fact, in Stan, you could replace the "sampling statement"

```
y ~ normal(mu,sigma)
```

with a corresponding command to increment the log-probability:

```
increment_log_prob( normal_log( y,mu,sigma ))
```

and get the same result. In the command above, `normal_log` is the logarithm of the normal density, and `increment_log_prob` is the function that adds the result to the running total of the log-probability.

The two Stan statements above, while yielding the same posterior MCMC sample, are not completely equivalent, however. The sampling statement assumes that all you want is a representative sample from the posterior distribution, for which all that is needed is the relative not absolute posterior density, so Stan cleverly removes all the constants from the formula for the normal density to improve efficiency in computation. The explicit log-probability form retains the exact posterior density.

One benefit of this computational method for Stan is that you have great flexibility in specifying whatever distribution you want for your model. All you need to do is express the log-probability inside the `increment_log_prob` function. At least, in principle. Stan must also be able to figure out a gradient for the Hamiltonian dynamics. My point of mentioning this here is for you to understand a bit better why log-probability and gradients are so central to the architecture of Stan. For advanced programming details, please see the Stan reference manual.

14.3.4. Sampling the prior in Stan

As was discussed in Section 8.5, p. 211, there are several reasons why we might want to examine a sample from the prior distribution of a model. This can be especially useful for viewing the implied prior on mid-level parameters in a hierarchical model, or for viewing the implied prior on derived parameters such as differences of means.

In the current version of Stan, data cannot have missing values. (Unlike JAGS, which imputes missing values as if they were parameters to be estimated.) Therefore the prior distribution cannot be sampled in Stan merely by commenting out the data as we did with JAGS in Section 8.5. Instead, we comment out the likelihood from the model specification. Here is an example:

```
modelString = "
  data {
    int<lower=0> N ;
    int y[N] ;
  }
  parameters {
    real<lower=0,upper=1> theta ;
  }
  model {
    theta ~ beta(1,1) ;
//    y ~ bernoulli(theta) ;  // likelihood commented out
  }
" # close quote for modelString
```

You must recompile the model with the `stan_model` command. Then sample from the model exactly as before, with the same data as before.

You can understand why this works by thinking about what the sampling statement (now commented out) really means. As mentioned in the previous section, the sampling statement is just incrementing the log-probability according to the data. If we leave that line out, then the log-probability is influenced only by the other statements in the model, namely, the statements that specify the prior.

Stan can have convergence problems when sampling from very diffuse, "flat" distributions because there is such a small gradient. If Stan has trouble sampling from the prior of your model, you can experiment with the priors to see if less

diffuse priors solve the convergence problem, and still get a sense of the qualitative nature of the implied priors on derived parameters. JAGS does not have this problem.

14.3.5. Simplified scripts for frequently used analyses

As you will fondly recall from Section 8.3, p. 206, I have wrapped the JAGS scripts for frequently used analyses in functions that can be called using a consistent sequence of commands across different analyses. That way, for example, the single command `plotMCMC` will display different sets of graphs for different applications. I have done the same wrapping for analogous Stan scripts. The file names start with "Stan-" instead of with "Jags-."

For example, the simple Bernoulli-beta model described above is called with the script named `Stan-Ydich-Xnom1subj-MbernBeta-Example.R`. The file name convention was explained in Section 8.3. The script is virtually identical to the JAGS version, except that "Jags" is replaced with "Stan" and there are a few extra lines at the end to illustrate the use of RStan plotting functions:

```
# Load The data
myData = read.csv("z15N50.csv")
# Load the functions genMCMC, smryMCMC, and plotMCMC:
source("Stan-Ydich-Xnom1subj-MbernBeta.R")
# Specify filename root and graphical format for saving output.
fileNameRoot = "Stan-Ydich-Xnom1subj-MbernBeta-"
graphFileType = "eps" # or "png" or "pdf" etc.
# Generate the MCMC chain:
mcmcCoda = genMCMC( data=myData , numSavedSteps=10000 , saveName=fileNameRoot )
# Display diagnostics of chain, for specified parameters:
parameterNames = varnames(mcmcCoda) # get all parameter names
for ( parName in parameterNames ) {
  diagMCMC( mcmcCoda , parName=parName ,
                saveName=fileNameRoot , saveType=graphFileType )
}
# Get summary statistics of chain:
summaryInfo = smryMCMC( mcmcCoda , compVal=0.5 , rope=c(0.45,0.55) ,
                        saveName=fileNameRoot )
# Display posterior information:
plotMCMC( mcmcCoda , data=myData , # compVal=0.5 , rope=c(0.45,0.55) ,
          saveName=fileNameRoot , saveType=graphFileType )
# Use Stan display functions instead of DBDA2E functions:
# Load the stanFit object that was saved by genMCMC:
load("Stan-Ydich-Xnom1subj-MbernBeta-StanFit.Rdata")
# Display information:
show(stanFit)
openGraph()
```

```
traceplot(stanFit,pars=c("theta"))
openGraph()
plot(stanFit,pars=c("theta"))
```

14.4. SPECIFY MODELS TOP-DOWN IN STAN

For humans, descriptive models begin, conceptually, with the data that are to be described. We first know the measurement scale of the data and their structure. Then we conceive of a likelihood function for the data. The likelihood function has meaningful parameters, which we might want to re-express in terms of other data (called covariates, predictors, or regressors). Then we build a meaningful hierarchical prior on the parameters. Finally, at the top level, we specify constants that express our prior knowledge, which might be vague or noncommittal.

A nice feature of JAGS is that models can be specified in that conceptual order: from data, to likelihood, to successively higher levels of the prior. This format was discussed in Section 8.2.2, p. 198, where it was also explained that JAGS does not care about the order in which the dependencies are specified in its model statement. JAGS does not execute the lines in order as a procedure; instead JAGS examines the specifications of the dependencies, checks them for consistency, and assembles MCMC samplers for the corresponding model structure.

This bottom-up ordering is *not* appropriate for model specifications in Stan, however. Stan translates the model specification directly into corresponding C++ commands, which are processed in order. For example, we cannot start a model specification with `y ~ bernoulli(theta)` because we have not yet told Stan the value of `theta`. Stan would try to fill in the value of `theta` with a default value from the variable's declaration or with the value from the previous MCMC step, neither of which is want we intend. Therefore, *in Stan, model specifications usually begin with the top level of the prior, and then lower-level dependencies are filled in, finishing with the likelihood of the data.*

The fact that Stan executes the model specification in order can be very useful for flow control in complex models. For example, unlike JAGS, Stan has `while` loops, which keep cycling until a condition has been met. Stan also has `if else` structures, not available in JAGS.

I find that when I am initially typing a new model in Stan ("typing" in the sense of manually poking fingers on a computer keyboard, not "typing" in the sense of declaring variable types), I type the model in bottom-up conceptual order, moving the cursor around to different blocks of the specification as I go. Before any typing, I make a (hand sketched) diagram of the model as in Figure 8.2, p. 196. Then I start by typing the likelihood function, such as `y ~ bernoulli(theta)` in the model block. Then I move the cursor to the data block and declare the `y` and `N` variables, then I move the cursor to the parameter block and declare the `theta` variable. Then I move, conceptually,

to the next level up the model structure, visually moving up the hierarchical diagram (as in Figure 8.2, p. 196). I place the cursor at the start of the model block, *above* the previously specified likelihood function, and type the corresponding specification, `theta ~ beta(1,1)`. If this statement had introduced new higher-level parameters or data, I would then declare them in the appropriate blocks. This process repeats until reaching the top of the hierarchical structure. After the model has been typed-in using bottom-up conceptual order, I then check the model specification for accuracy and efficiency in procedural order, reading the text from top to bottom. I ask myself, Are variables processed in correct logical order? Are loops and if-else statements really in the correct order? Could the processing efficiency be improved by using different flow structure or vectorization?

14.5. LIMITATIONS AND EXTRAS

At the time of this writing, one of the main limitations of Stan is that it does not allow discrete (i.e., categorical) parameters. The reason for this limitation is that Stan has HMC as its foundational sampling method, and HMC requires computing the gradient (i.e., derivative) of the posterior distribution with respect to the parameters. Of course, gradients are undefined for discrete parameters. The Stan 2.1.0 reference manual states, "Plans are in place to add full discrete sampling in Stan 2.0" (p. 4, Footnote 2). Perhaps by the time you read this book, Stan will allow discrete parameters.

In particular, the lack of discrete parameters in Stan means that we cannot do model comparison as a hierarchical model with an indexical parameter at the top level, as was done in Section 10.3.2. There might be ways to work around this restriction by using clever programming contrivances, but presently there is nothing as straight forward as the model specification in JAGS.

A facility of Stan that is not present in JAGS is the ability to find the mode of the posterior distribution in the joint parameter space, directly from the formulation of the model and not from an MCMC sample. If the posterior distribution is curved, the multidimensional mode is not necessarily the same as the marginal modes. We will not use this facility for the applications in this book, however.

14.6. EXERCISES

Look for more exercises at https://sites.google.com/site/doingbayesiandataanalysis/

Exercise 14.1. [Purpose: Transformed parameters in Stan, and comparison with JAGS.] For this exercise, we analyze in Stan the therapeutic-touch data from Section 9.2.4, p. 240. The model structure is depicted in Figure 9.7, p. 236. The relevant Stan scripts are included in the online suite of programs that accompany the book; see

`Stan-Ydich-XnomSsubj-MbernBetaOmegaKappa-Example.R`. Compare it with the JAGS version, which is the same file name but starting with Jags-.

(A) Consider the model block of the Stan code called by the script. What does the `transformed parameters` block do?

(B) Consider the model block of the Stan code called by the script. Why is `theta` vectorized but not `y`?

(C) Run the Stan program, and note how long it takes. Run the Jags version and note how long it takes. Do they produce the same posterior distribution with the same effective sample size (ESS)? Which is faster? You might find that Stan is no faster, or even much slower. Thus, there is no guarantee that Stan will work better. Please note that to make comparisons between Stan and Jags, the ESS of the resulting chains must be matched, not the number of steps. The summaries and graphs created by the `coda`-based functions are convenient for displaying the ESS computed with the same ESS algorithm for the outputs of both Stan and JAGS. For a fair comparison of Stan and JAGS, they also must be matched for their required burn-in phases, as they might need different amounts of burn-in to reach converged chains.

Exercise 14.2. [Purpose: Power analysis in Stan.] It was briefly mentioned in Section 14.3.1 that the DSO created by the Stan command `stan_model` could be reused in multiple calls of the Stan `sampling` command. For this exercise, your goal is to implement in Stan the Monte Carlo approximation of power described in Section 13.2.4, p. 372. It is straight forward to implement it in Stan using the exactly analogous function definitions, but this approach, while logically correct, is inefficient because it will recompile the Stan model every time a new simulated data set is analyzed. The real exercise of this exercise is to modify the function that does the Stan analysis so that it can either compile the Stan model afresh or reuse an existing DSO. In your modified version, *compile the Stan model outside the loop that runs the analyses on simulated data, and pass the compiled DSO as an argument into the function that runs an analysis on a simulated data set*. Show your code, with explanatory comments.