are simulated, one for each density q_k in the ladder. Each chain moves on its own but with occasional flipping of states between chains, with a Metropolis accept-reject rule similar to that in simulated tempering. At convergence, the simulations from chain 0 represent draws from the target distribution.

Other auxiliary variable methods have been developed that are tailored to particular structures of multivariate distributions. For example, highly correlated variables such as arise in spatial statistics can be simulated using *multigrid sampling*, in which computations are done alternately on the original scale and on coarser scales that do not capture the local details of the target distribution but allow faster movement between states.

Particle filtering, weighting, and genetic algorithms

Particle filtering describes a class of simulation algorithms involving parallel chains, in which existing chains are periodically tested and allowed to die, live, or split, with the rule set up so that chains in lower-probability areas of the posterior distribution are more likely to die and those in higher-probability areas are more likely to split. The idea is that a large number of chains can explore the parameter space, with the birth/death/splitting steps allowing the ensemble of chains to more rapidly converge to the target distribution. The probabilities of the different steps are set up so that the stationary distribution of the entire process is the posterior distribution of interest.

A related idea is *weighting*, in which a simulation is performed that converges to a specified but wrong distribution, $g(\theta)$, and then the final draws are weighted by $p(\theta|y)/g(\theta)$. In more sophisticated implementations, this reweighting can be done throughout the simulation process. It can sometimes be difficult or expensive to sample from $p(\theta|y)$ and faster to work with a good approximation g if available. Weighting can be combined with particle filtering by using the weights in the die/live/split probabilities.

Genetic algorithms are similar to particle filtering in having multiple chains that can live or die, but with the elaboration that the updating algorithms themselves can change ('mutate') and combine ('sexual reproduction'). Many of these ideas are borrowed from the numerical analysis literature on optimization but can also be effective in a posterior simulation setting in which the goal is to converge to a distribution rather than to a single best value.

12.4 Hamiltonian Monte Carlo

An inherent inefficiency in the Gibbs sampler and Metropolis algorithm is their random walk behavior—as illustrated in Figures 11.1 and 11.2 on pages 276 and 277, the simulations can take a long time zigging and zagging while moving through the target distribution. Reparameterization and efficient jumping rules can improve the situation (see Sections 12.1 and 12.2), but for complicated models this local random walk behavior remains, especially for high-dimensional target distributions.

Hamiltonian Monte Carlo (HMC) borrows an idea from physics to suppress the local random walk behavior in the Metropolis algorithm, thus allowing it to move much more rapidly through the target distribution. For each component θ_j in the target space, Hamiltonian Monte Carlo adds a 'momentum' variable ϕ_j. Both θ and ϕ are then updated together in a new Metropolis algorithm, in which the jumping distribution for θ is determined largely by ϕ. Each iteration of HMC proceeds via several steps, during which the position and momentum evolve based on rules imitating the behavior of position the steps can move rapidly where possible through the space of θ and even can turn corners in parameter space to preserve the total 'energy' of the trajectory. Hamiltonian Monte Carlo is also called *hybrid Monte Carlo* because it combines MCMC and deterministic simulation methods.

In HMC, the posterior density $p(\theta|y)$ (which, as usual, needs only be computed up

to a multiplicative constant) is augmented by an independent distribution $p(\phi)$ on the momenta, thus defining a joint distribution, $p(\theta, \phi|y) = p(\phi)p(\theta|y)$. We simulate from the joint distribution but we are only interested in the simulations of θ; the vector ϕ is thus an auxiliary variable, introduced only to enable the algorithm to move faster through the parameter space.

In addition to the posterior density (which, as usual, needs to be computed only up to a multiplicative constant), HMC also requires the gradient of the log-posterior density. In practice the gradient must be computed analytically; numerical differentiation requires too many function evaluations to be computationally effective. If θ has d dimensions, this gradient is $\frac{d \log p(\theta|y)}{d\theta} = \left(\frac{d \log p(\theta|y)}{d\theta_1}, \ldots, \frac{d \log p(\theta|y)}{d\theta_d} \right)$. For most of the models we consider in this book, this vector is easy to determine analytically and then program. When writing and debugging the program, we recommend also programming the gradient numerically (using finite differences of the log-posterior density) as a check on the programming of the analytic gradients. If the two subroutines do not return identical results to several decimal places, there is likely a mistake somewhere.

The momentum distribution, $p(\phi)$

It is usual to give ϕ a multivariate normal distribution (recall that ϕ has the same dimension as θ) with mean 0 and covariance set to a prespecified 'mass matrix' M (so called by analogy to the physical model of Hamiltonian dynamics). To keep it simple, we commonly use a diagonal mass matrix, M. If so, the components of ϕ are independent, with $\phi_j \sim \text{N}(0, M_{jj})$ for each dimension $j = 1, \ldots, d$. It can be useful for M to roughly scale with the inverse covariance matrix of the posterior distribution, $(\text{var}(\theta|y))^{-1}$, but the algorithm works in any case; better scaling of M will merely make HMC more efficient.

The three steps of an HMC iteration

HMC proceeds by a series of iterations (as in any Metropolis algorithm), with each iteration having three parts:

1. The iteration begins by updating ϕ with a random draw from its posterior distribution—which, as specified, is the same as its prior distribution, $\phi \sim \text{N}(0, M)$.
2. The main part of the Hamiltonian Monte Carlo iteration is a simultaneous update of (θ, ϕ), conducted in an elaborate but effective fashion via a discrete mimicking of physical dynamics. This update involves L 'leapfrog steps' (to be defined in a moment), each scaled by a factor ϵ. In a leapfrog step, both θ and ϕ are changed, each in relation to the other. The L leapfrog steps proceed as follows:
   Repeat the following steps L times:
   (a) Use the gradient (the vector derivative) of the log-posterior density of θ to make a half-step of ϕ:
   $$\phi \leftarrow \phi + \frac{1}{2}\epsilon \frac{d \log p(\theta|y)}{d\theta}.$$
   (b) Use the 'momentum' vector ϕ to update the 'position' vector θ:
   $$\theta \leftarrow \theta + \epsilon M^{-1}\phi.$$
   Again, M is the mass matrix, the covariance of the momentum distribution $p(\phi)$. If M is diagonal, the above step amounts to scaling each dimension of the θ update. (It might seem redundant to include ϵ in the above expression: why not simply absorb it into M, which can itself be set by the user? The reason is that it can be convenient in tuning the algorithm to alter ϵ while keeping M fixed.)

(c) Again use the gradient of θ to half-update ϕ:

$$\phi \leftarrow \phi + \frac{1}{2}\epsilon \frac{d \log p(\theta|y)}{d\theta}.$$

Except at the first and last step, updates (c) and (a) above can be performed together. The stepping thus starts with a half-step of ϕ, then alternates $L-1$ full steps of the parameter vector θ and the momentum vector ϕ, and concludes with a half-step of ϕ. This algorithm (called a 'leapfrog' because of the splitting of the momentum updates into half steps) is a discrete approximation to physical Hamiltonian dynamics in which both position and momentum evolve in continuous time.

In the limit of ϵ near zero, the leapfrog algorithm preserves the joint density $p(\theta, \phi|y)$. We will not give the proof, but here is some intuition. Suppose the current value of θ is at a flat area of the posterior. Then $\frac{d \log p(\theta|y)}{d\theta}$ will be zero, and in step 2 above, the momentum will remain constant. Thus the leapfrog steps will skate along in θ-space with constant velocity. Now suppose the algorithm moves toward an area of low posterior density. Then $\frac{d \log p(\theta|y)}{d\theta}$ will be negative in this direction, thus in step 2 inducing a decrease in the momentum in the direction of movement. As the leapfrog steps continue to move into an area of lower density in θ-space, the momentum continues to decrease. The decrease in $\log p(\theta|y)$ is matched (in the limit $\epsilon \to 0$, exactly so) by a decrease in the 'kinetic energy,' $\log p(\phi)$. And if iterations continue to move in the direction of decreasing density, the leapfrog steps will slow to zero and then back down or curve around the dip. Now consider the algorithm heading in a direction in which the posterior density is increasing. Then $\frac{d \log p(\theta|y)}{d\theta}$ will be positive in that direction, leading in step 2 to an increase in momentum in that direction. As $\log p(\theta|y)$ increases, $\log p(\phi)$ increases correspondingly until the trajectory eventually moves past or around the mode and then starts to slow down.

For finite ϵ, the joint density $p(\theta, \phi|y)$ does not remain entirely constant during the leapfrog steps but it will vary only slowly if ϵ is small. For reasons we do not discuss here, the leapfrog integrator has the pleasant property that combining L steps of error δ does not produce $L\delta$ error, because the dynamics of the algorithm tend to send the errors weaving back and forth around the exact value that would be obtained by a continuous integration. Keeping the discretization error low is important because of the next part of the HMC algorithm, the accept/reject step.

3. Label θ^{t-1}, ϕ^{t-1} as the value of the parameter and momentum vectors at the start of the leapfrog process and θ^*, ϕ^* as the value after the L steps. In the accept-reject step, we compute

$$r = \frac{p(\theta^*|y)p(\phi^*)}{p(\theta^{t-1}|y)p(\phi^{t-1})}. \tag{12.3}$$

4. Set

$$\theta^t = \begin{cases} \theta^* & \text{with probability } \min(r, 1) \\ \theta^{t-1} & \text{otherwise.} \end{cases}$$

Strictly speaking it would be necessary to set ϕ^t as well, but since we do not care about ϕ in itself, and it gets immediately updated at the beginning of the next iteration (see step 1 above), so there is no need to keep track of it after the accept/reject step.

As with any other MCMC algorithm, we repeat these iterations until approximate convergence, as assessed by $\hat{R}$ being near 1 and the effective sample size being large enough for all quantities of interest; see Section 11.4.

Restricted parameters and areas of zero posterior density

HMC is designed to work with all-positive target densities. If at any point during an iteration the algorithm reaches a point of zero posterior density (for example, if the steps go below zero when updating a parameter that is restricted to be positive), we stop the stepping and give up, spending another iteration at the previous value of θ. The resulting algorithm preserves detailed balance and stays in the positive zone.

An alternative is 'bouncing,' where again the algorithm checks that the density is positive after each step and, if not, changes the sign of the momentum to return to the direction in which it came. This again preserves detailed balance and is typically more efficient than simply rejecting the iteration, for example with a hard boundary for a parameter that is restricted to be positive.

Another way to handle bounded parameters is via transformation, for example taking the logarithm of a parameter constrained to be positive or the logit for a parameter constrained to fall beween 0 and 1, or more complicated joint transformations for sets of parameters that are constrained (for example, if $\theta_1 < \theta_2 < \theta_3$ or if $\alpha_1 + \alpha_2 + \alpha_3 + \alpha_4 = 1$). One must then work out the Jacobian of the transformation and use it to determine the log posterior density and its gradient in the new space.

Setting the tuning parameters

HMC can be tuned in three places: (i) the probability distribution for the momentum variables ϕ (which, in our implementation requires specifying the diagonal elements of a covariance matrix, that is, a scale parameter for each of the d dimensions of the parameter vector), (ii) the scaling factor ϵ of the leapfrog steps, and (iii) the number of leapfrog steps L per iteration.

As with the Metropolis algorithm in general, these tuning parameters can be set ahead of time, or they can be altered completely at random (a strategy which can sometimes be helpful in keeping an algorithm from getting stuck), but one has to take care when altering them given information from previous iterations. Except in some special cases, adaptive updating of the tuning parameters alters the algorithm so that it no longer converges to the target distribution. So when we set the tuning parameters, we do so during the warm-up period: that is, we start with some initial settings, then run HMC for a while, then reset the tuning parameters based on the iterations so far, then discard the early iterations that were used for warm-up. This procedure can be repeated if necessary, as long as the saved iterations use only simulations after the last setting of the tuning parameters.

How, then, to set the parameters that govern HMC? We start by setting the scale parameters for the momentum variables to some crude estimate of the scale of the target distribution. (One can also incorporate covariance information but here we will assume a diagonal covariance matrix so that all that is required is the vector of scales.) By default we could simply use the identity matrix.

We then set the product ϵL to 1. This roughly calibrates the HMC algorithm to the 'radius' of the target distribution; that is, L steps, each of length ϵ times the already-chosen scale of ϕ, should roughly take you from one side of the distribution to the other. A default starting point could be $\epsilon = 0.1, L = 10$.

Finally, theory suggests that HMC is optimally efficient when its acceptance rate is approximately 65% (based on an analysis similar to that which finds an optimal 23% acceptance rate for the multidimensional Metropolis algorithm). The theory is based on all sorts of assumptions but seems like a reasonable guideline for optimization in practice. For now we recommend a simple adaptation in which HMC is with its initial settings and then adapted if the average acceptance probability (as computed from the simulations so far) is not close to 65%. If the average acceptance probability is lower, then the leapfrog jumps

are too ambitious and you should lower ϵ and correspondingly increase L (so their product remains 1). Conversely, if the average acceptance probability is much higher than 65%, then the steps are too cautious and we recommend raising ϵ and lowering L (not forgetting that L must be an integer). These rules do not solve all problems, and it should be possible to develop diagnostics to assess the efficiency of HMC to allow for more effective adaptation of the tuning parameters.

Varying the tuning parameters during the run

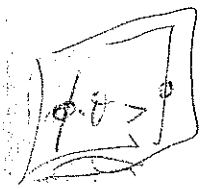
As with MCMC tuning more generally, any adaptation can go on during the warm-up period, but adaptation performed later on, during the simulations that will be used for inference, can cause the algorithm to converge to the wrong distribution. For example, suppose we were to increase the step size ϵ after high-probability jumps and decrease ϵ when the acceptance probability is low. Such an adaptation seems appealing but would destroy the detailed balance (that is, the property of the algorithm that the flow of probability mass from point A to B is the same as from B to A, for any points A and B in the posterior distribution) that is used to prove that the posterior distribution of interest is the stationary distribution of the Markov chain.

Completely random variation of ϵ and L, however, causes no problems with convergence and can be useful. If we randomly vary the tuning parameters (within specified ranges) from iteration to iteration while the simulation is running, the algorithm has a chance to take long tours through the posterior distribution when possible and make short movements where the iterations are stuck in a cramped part of the space. The price for this variation is some potential loss of optimality, as the algorithm will also take short steps where long tours would be feasible and try for long steps where the space is too cramped for such jumps to be accepted.

Locally adaptive HMC

For difficult HMC problems, it would be desirable for the tuning parameters to vary as the algorithm moves through the posterior distribution, with the mass matrix M scaling to the local curvature of the log density, the step size ϵ getting smaller in areas where the curvature is high, and the number of steps L being large enough for the trajectory to move far through the posterior distribution without being so large that the algorithm circles around and around. To this end, researchers have developed extensions of HMC that adapt without losing detailed balance. These algorithms are more complicated and can require more computations per iteration but can converge more effectively for complicated distributions. We describe two such algorithms here but without giving the details.

The no-U-turn sampler. In the no-U-turn sampler, the number of steps is determined adaptively at each iteration. Instead of running for a fixed number of steps, L, the trajectory in each iteration continues until it turns around (more specifically, until we reach a negative value of the dot product between the momentum variable ϕ and the distance traveled from the position θ at the start of the iteration). This rule essentially sends the trajectory as far as it can go during that iteration. If such a rule is applied alone, the simulations will not converge to the desired target distribution. The full no-U-turn sampler is more complicated, going backward and forward along the trajectory in a way that satisfies detailed balance. Along with this algorithm comes a procedure for adaptively setting the mass matrix M and step size ϵ; these parameters are tuned during the warm-up phase and then held fixed during the later iterations which are kept for the purpose of posterior inference.



Riemannian adaptation. Another approach to optimization is Riemannian adaptation, in which the mass matrix M is set to conform with the local curvature of the log posterior

density at each step. Again, the local adaptation allows the sampler to move much more effectively but the steps of the algorithm need to become more complicated to maintain detailed balance. Riemannian adaptation can be combined with the no-U-turn sampler.

Neither of the above extensions solves all the problems with HMC. The no-U-turn sampler is self-tuning and computationally efficient but, like ordinary Hamiltonian Monte Carlo, has difficulties with very short-tailed and long-tailed distributions, in both cases having difficulties transitioning from the center to the tails, even in one dimension. Riemannian adaptation handles varying curvature and non-exponentially tailed distributions but is impractical in high dimensions.

Combining HMC with Gibbs sampling

There are two ways in which ideas of the Gibbs sampler fit into Hamiltonian Monte Carlo. First, it can make sense to partition variables into blocks, either to simplify computation or to speed convergence. Consider a hierarchical model with J groups, with parameter vector $\theta = (\eta^{(1)}, \eta^{(2)}, \ldots, \eta^{(J)}, \phi)$, where each of the $\eta^{(j)}$'s is itself a vector of parameters corresponding to the model for group j and ϕ is a vector of hyperparameters, and for which the posterior distribution can be factored as, $p(\theta|y) \propto p(\phi) \prod_{j=1}^{J} p(\eta^{(j)}|\phi) p(y^{(j)}|\eta^{(j)})$. In this case, even if it is possible to update the entire vector θ at once using HMC, it may be more effective—in computation speed or convergence—to cycle through $J+1$ updating steps, altering each $\eta^{(j)}$ and then ϕ during each cycle. This way we only have to work with at most one of the likelihood factors, $p(y^{(j)}|\eta^{(j)})$, at each step. Parameter expansion can be used to facilitate quicker mixing through the joint distribution.

The second way in which Gibbs sampler principles can enter HMC is through the updating of discrete variables. Hamiltonian dynamics are only defined on continuous distributions. If some of the parameters in a model are defined on discrete spaces (for example, latent indicators for mixture components, or a parameter that follows a continuous distribution but has a positive probability of being exactly zero), they can be updated using Gibbs steps or, more generally, one-dimensional updates such as Metropolis or slice sampling (see Section 12.3). The simplest approach is to partition the space into discrete and continuous parameters, then alternate HMC updates on the continuous subspace and Gibbs, Metropolis, or slice updates on the discrete components.

12.5 Hamiltonian dynamics for a simple hierarchical model

We illustrate the tuning of Hamiltonian Monte Carlo with the model for the educational testing experiments described in Chapter 5. HMC is not necessary in this problem—the Gibbs sampler works just fine, especially after the parameter expansion which allows more efficient movement of the hierarchical variance parameter (see Section 12.1)—but it is helpful to understand the new algorithm in a simple example. Here we go through all the steps of the algorithm. The code appears in Section C.4, starting on page 601.

In order not to overload our notation, we label the eight school effects (defined as θ_j in Chapter 5) as α_j; the full vector of parameters θ then has $d = 10$ dimensions, corresponding to $\alpha_1, \ldots, \alpha_8, \mu, \tau$.

Gradients of the log posterior density. For HMC we need the gradients of the log posterior density for each of the ten parameters, a set of operations that are easily performed with the normal distributions of this model:

$$\frac{d \log p(\theta|y)}{d\alpha_j} = -\frac{\alpha_j - y_j}{\sigma_j^2} - \frac{\alpha_j - \mu}{\tau^2}, \text{ for } j = 1, \ldots, 8,$$

$$\frac{d \log p(\theta|y)}{d\mu} = -\sum_{j=1}^{J} \frac{\mu - \alpha_j}{\tau^2},$$

$$\frac{d \log p(\theta|y)}{d\tau} = -\frac{J}{\tau} + \sum_{j=1}^{J} \frac{(\mu - \alpha_j)^2}{\tau^3}.$$

As a debugging step we also compute the gradients numerically using finite differences of ± 0.0001 on each component of θ. Once we have checked that the two gradient routines yield identical results, we use the analytic gradient in the algorithm as it is faster to compute.

The mass matrix for the momentum distribution. As noted above, we want to scale the mass matrix to roughly match the posterior distribution. That said, we typically only have a vague idea of the posterior scale before beginning our computation; thus this scaling is primarily intended to forestall the problems that would arise if there are gross disparities in the scaling of different dimensions. In this case, after looking at the data in Table 5.2 we assign a rough scale of 15 for each of the parameters in the model and crudely set the mass matrix to $\text{Diag}(15, \ldots, 15)$.

Starting values. We run 4 chains of HMC with starting values drawn at random to crudely match the scale of the parameter space, in this case following the idea above and drawing the ten parameters in the model from independent $N(0, 15^2)$ distributions.

Tuning ϵ and L. To give the algorithm more flexibility, we do not set ϵ and L to fixed values. Instead we choose central values ϵ_0, L_0 and then at each step draw ϵ and L independently from uniform distributions on $(0, 2\epsilon_0)$ and $[1, 2L_0]$, respectively (with the distribution for L being discrete uniform, as L must be an integer). We have no reason to think this particular jittering is ideal; it is just a simple way to vary the tuning parameters in a way that does not interfere with convergence of the algorithm. Following the general advice given above, we start by setting $\epsilon_0 L_0 = 1$ and $L_0 = 10$. We simulate 4 chains for 20 iterations just to check that the program runs without crashing.

We then do some experimentation. We first run 4 chains for 100 iterations and see that the inferences are reasonable (no extreme values, as can sometimes happen when there is poor convergence or a bug in the program) but not yet close to convergence, with several values of $\widehat{R}$ that are more than 2. The average acceptance probabilities of the 4 chains are 0.23, 0.59, 0.02, and 0.57, well below 65%, so we suspect the step size is too large.

We decrease ϵ_0 to 0.05, increase L_0 to 20 (thus keeping $\epsilon_0 L_0$ constant), and rerun the 4 chains for 100 iterations, now getting acceptance rates of 0.72,, 0.87, 0.33, and 0.55, with chains still far from mixing. At this point we increase the number of simulations to 1000. The simulations now are close to convergence, with $\widehat{R}$ less than 1.2 for all parameters, and average acceptance probabilities are more stable, at 0.52, 0.68, 0.75, and 0.51. We then run 4 chains at 10,000 simulations at these tuning parameters and achieve approximate convergence, with $\widehat{R}$ less than 1.1 for all parameters.

In this particular example, HMC is unnecessary, as the Gibbs sampler works fine on an appropriately transformed scale. In larger and more difficult problems, however, Gibbs and Metropolis can be too slow, while HMC can move effectively efficiently move through high-dimensional parameter spaces.

Transforming to $\log \tau$

When running HMC on a model with constrained parameters, the algorithm can go outside the boundary, thus wasting some iterations. One remedy is to transform the space to be unconstrained. In this case, the simplest way to handle the constraint $\tau > 0$ is to transform to $\log \tau$. We then must alter the algorithm in the following ways:

1. We redefine θ as $(\alpha_1, \ldots, \alpha_8, \mu, \log\tau)$ and do all jumping on this new space.
2. The (unnormalized) posterior density $p(\theta|y)$ is multiplied by the Jacobian, τ, so we add $\log\tau$ to the log posterior density used in the calculations.
3. The gradient of the log posterior density changes in two ways: first, we need to account for the new term added just above; second, the derivative for the last component of the gradient is now with respect to $\log\tau$ rather than τ and so must be multiplied by the Jacobian, τ:
$$\frac{d\log p(\theta|y)}{d\log\tau} = -(J-1) + \sum_{j=1}^{J}\frac{(\mu-\alpha_j)^2}{\tau^2}.$$
4. We change the mass matrix to account for the transformation. We keep $\alpha_1, \ldots, \alpha_8, \mu$ with masses of 15 (roughly corresponding to a posterior distribution with a scale of 15 in each of these dimensions) but set the mass of $\log\tau$ to 1.
5. We correspondingly change the initial values by drawing the first nine parameters from independent $\mathrm{N}(0, 15^2)$ distributions and $\log\tau$ from $\mathrm{N}(0, 1)$.

HMC runs as before. Again, we start with $\epsilon = 0.1$ and $L = 10$ and then adjust to get a reasonable acceptance rate.

12.6 Stan: developing a computing environment

Hamiltonian Monte Carlo takes a bit of effort to program and tune. In more complicated settings, though, we have found HMC to be faster and more reliable than basic Markov chain simulation algorithms.

To mitigate the challenges of programming and tuning, we have developed a computer program, Stan (Sampling through adaptive neighborhoods) to automatically apply HMC given a Bayesian model. The key steps of the algorithm are data and model input, computation of the log posterior density (up to an arbitrary constant that cannot depend on the parameters in the model) and its gradients, a warm-up phase in which the tuning parameters are set, an implementation of the no-U-turn sampler to move through the parameter space, and convergence monitoring and inferential summaries at the end.

We briefly describe how each of these steps is done in Stan. Instructions and examples for running the program appear in Appendix C.

Entering the data and model

Each line of a Stan model goes into defining the log probability density of the data and parameters, with code for looping, conditioning, computation of intermediate quantities, and specification of terms of the log joint density. Standard distributions such as the normal, gamma, binomial, Poisson, and so forth, are preprogrammed, and arbitrary distributions can be entered by directly programming the log density. Algebraic manipulations and functions such as exp and logit can also be included in the specification; it is all just sent into C++.

To compute gradients, Stan uses automatic analytic differentiation, using an algorithm that parses arbitrary C++ expressions and then applies basic rules of differential calculus to construct a C++ program for the gradient. For computational efficiency, we have preprogrammed the gradients for various standard statistical expressions to make up some of this difference. We use special scalar variable classes that evaluate the function and at the same time construct the full expression tree used to generate the log probability. Then the reverse pass walks backward down the expression tree (visiting every dependent node before any node it depends on), propagating partial derivatives by the chain rule. The walk over the expression tree implicitly employs dynamic programming to minimize the number of

calculations. The resulting autodifferentiation is typically much faster than computing the gradient numerically via finite differences.

In addition to the data, parameters, and model statements, a Stan call also needs the number of chains, the number of iterations per chain, and various control parameters that can be set by default. Starting values can be supplied or else they are generated from preset default random variables.

Setting tuning parameters in the warm-up phase

As noted above, it can be tricky to tune Hamiltonian Monte Carlo for any particular example. The no-U-turn sampler helps with this, as it eliminates the need to assign the number of steps L, but we still need to set the mass matrix M and step size ϵ. During a prespecified warm-up phase of the simulation, Stan adaptively alters M and ϵ using ideas from stochastic optimization in numerical analysis. This adaptation will not always work—for distributions with varying curvature, there will not in general be any single good set of tuning parameters—and if the simulation is having difficulty converging, it can make sense to look at the values of M and ϵ chosen for different chains to better understand what is happening. Convergence can sometimes be improved by reparameterization. More generally, it could make sense to have different tuning parameters for different areas of the distribution—this is related to ideas such as Riemannian adaptation, which at the time of this writing we are incorporating into Stan.

No-U-turn sampler

Stan runs HMC using the no-U-turn sampler, preprocessing where possible by transforming bounded variables to put them on an unconstrained scale. For complicated constraints this cannot always be done automatically and then it can make sense for the user to reparameterize in writing the model. While running, Stan keeps track of acceptance probabilities (as well as the simulations themselves), which can be helpful in getting inside the algorithm if there are problems with mixing of the chains.

Inferences and postprocessing

Stan produces multiple sequences of simulations. For our posterior inferences we discard the iterations from the warm-up period (but we save them as possibly of diagnostic use if the algorithm is not mixing well) and compute $\widehat{R}$ and n_{eff} as described in Section 11.4.

12.7 Bibliographic note

For the relatively simple ways of improving simulation algorithms mentioned in Sections 12.1 and 12.2, Tanner and Wong (1987) discuss data augmentation and auxiliary variables, and Hills and Smith (1992) and Roberts and Sahu (1997) discuss different parameterizations for the Gibbs sampler. Higdon (1998) discusses some more complicated auxiliary variable methods, and Liu and Wu (1999), van Dyk and Meng (2001), and Liu (2003) present different approaches to parameter expansion. The results on acceptance rates for efficient Metropolis jumping rules appear in Gelman, Roberts, and Gilks (1995); more general results for Metropolis-Hastings algorithms appear in Roberts and Rosenthal (2001) and Brooks, Giudici, and Roberts (2003).

Gelfand and Sahu (1994) discuss the difficulties of maintaining convergence to the target distribution when adapting Markov chain simulations, as discussed at the end of Section 12.2. Andrieu and Robert (2001) and Andrieu and Thoms (2008) consider adaptive Markov chain Monte Carlo algorithms.

Slice sampling is discussed by Neal (2003), and simulated tempering is discussed by Geyer and Thompson (1993) and Neal (1996b). Besag et al. (1995) and Higdon (1998) review several ideas based on auxiliary variables that have been useful in high-dimensional problems arising in genetics and spatial models.

Reversible jump MCMC was introduced by Green (1995); see also Richardson and Green (1997) and Brooks, Giudici, and Roberts (2003) for more on trans-dimensional MCMC.

Mykland, Tierney, and Yu (1994) discuss an approach to MCMC in which the algorithm has regeneration points, or subspaces of θ, so that if a finite sequence starts and ends at a regeneration point, it can be considered as an exact (although dependent) sample from the target distribution. Propp and Wilson (1996) and Fill (1998) introduce a class of MCMC algorithms called perfect simulation in which, after a certain number of iterations, the simulations are known to have exactly converged to the target distribution.

The book by Liu (2001) covers a wide range of advanced simulation algorithms including those discussed in this chapter. The monograph by Neal (1993) also overviews many of these methods. Hamiltonian Monte Carlo was introduced by Duane et al. (1987) in the physics literature and Neal (1994) for statistics problems. Neal (2011) reviews HMC, Hoffman and Gelman (2013) introduce the no-U-turn sampler, and Girolami and Calderhead (2011) introduce Riemannian updating; see also Betancourt and Stein (2011) and Betancourt (2012, 2013). Romeel (2011) explains how leapfrog steps tend to reduce discretization error in HMC. Leimkuhler and Reich (2004) discuss the mathematics in more detail. Griewank and Walther (2008) is a standard reference on algorithmic differentiation.

12.8 Exercises

1. Efficient Metropolis jumping rules: Repeat the computation for Exercise 11.2 using the adaptive algorithm given in Section 12.2.
2. Simulated tempering: Consider the Cauchy model, $y_i \sim \text{Cauchy}(\theta, 1), i = 1, \ldots, n$, with two data points, $y_1 = 1.3, y_2 = 15.0$.
   (a) Graph the posterior density.
   (b) Program the Metropolis algorithm for this problem using a symmetric Cauchy jumping distribution. Tune the scale parameter of the jumping distribution appropriately.
   (c) Program simulated tempering with a ladder of 10 inverse-temperatures, 0.1, ..., 1.
   (d) Compare your answers in (b) and (c) to the graph in (a).
3. Hamiltonian Monte Carlo: Program HMC in R for the bioassay logistic regression example from Chapter 3.
   (a) Code the gradients analytically and numerically and check that the two programs give the same result.
   (b) Pick reasonable starting values for the mass matrix, step size, and number of steps.
   (c) Tune the algorithm to an approximate 65% acceptance rate.
   (d) Run 4 chains long enough so that each has an effective sample size of at least 100. How many iterations did you need?
   (e) Check that your inferences are consistent with those from the direct approach in Chapter 3.
4. Coverage of intervals and rejection sampling: Consider the following model: $y_j \sim \text{Binomial}(n_j, \theta_j)$, where $\theta_j = \text{logit}^{-1}(\alpha + \beta x_j)$, for $j = 1, \ldots, J$, and with independent prior distributions, $\alpha \sim t_4(0, 2^2)$ and $\beta \sim t_4(0, 1)$. Assume $J = 10$, the x_j values are randomly drawn from a U(1, 1) distribution, and $n_j \sim \text{Poisson}^+(5)$, where Poisson^+ is the Poisson distribution restricted to positive values.