8 Monte Carlo methods for inference

Monte Carlo methods are essential for sampling distributions and for estimating integrals, both of which lie at the heart of probabilistic inference. I start this chapter with a discussion of why sampling is non-trivial, and will then summarize where we use integration in inference and how we can estimate such integrals. We shall then look at the basic concepts of Monte Carlo sampling, and learn about a widely used Markov Chain Monte Carlo method: the Metropolis–Hastings algorithm.

8.1 Why we need efficient sampling

To recap section 3.3, the central equation in Bayesian inference tells us that the posterior PDF over the model parameters (θ) of a model (M) given the data (D) is the product of the likelihood and prior divided by the evidence

$$P(\theta\,|\,D, M) = \frac{P(D\,|\,\theta, M)P(\theta\,|\,M)}{P(D\,|\,M)}. \tag{8.1}$$

As the denominator is independent of the model parameters, we don't need it if we just want to find the shape of the posterior. In that case we can work with the unnormalized posterior,

$$P^*(\theta\,|\,D, M) = P(D\,|\,\theta, M)P(\theta\,|\,M). \tag{8.2}$$

With just one parameter, the posterior is a one-dimensional PDF. If appropriate, we can choose a conjugate prior for the given likelihood, in which case the posterior has the same functional form as the prior (see section 5.4). With the coin tossing example from section 5.1 the likelihood was binomial and the prior and posterior were beta distributions. If the posterior is not a standard distribution we could still evaluate it on a dense grid, as we did in sections 3.5 and 5.1 in one dimension and in section 6.3 in two dimensions. We also used these grid evaluations to integrate the distribution and thus to find the properly normalized posterior.

Evaluating on a grid is not as straightforward as it first seems, for a number of reasons. We must ensure that the grid covers the full range of the posterior. If the distribution has infinite support, we could use a grid only if we cover all the high probability density regions. Yet if the expression for the posterior is complicated, it will not be easy to identify these regions. And even if we can, there may still be a lot of probability contained in extended, low density tails. These cannot be neglected if we want to calculate any quantity involving the integral of the posterior, such as the mean or median. The posterior could also be highly peaked, in which case we need to use a very fine sampling around this region. The posterior could even be multimodal, so just sampling around the first maximum found will be insufficient. We could attempt to work out the optimal sampling range and density by trial and error: visualize the posterior and study whether quantities like the mean or mode are stable with changes in the sampling. But this is not a realistic option if we are processing a large number of data sets.

For problems with three or more parameters the situation is even more complicated. All of the above issues apply, but now the posterior exists in a higher dimensional parameter space, where more complex surfaces are possible that are impossible to visualize completely. It is also much more time consuming to determine the posterior by evaluating it on a regular, but now multi-dimensional, parameter grid. This is because of the curse of dimensionality. If we need N grid points in order to sample one parameter with sufficient density, then for two parameters we will need N^2 grid points. For J-dimensions we will need N^J grid points. Even with $J = 5$ and $N = 100$ (quite modest), we already need 10^{10} grid points, an infeasible number of calculations of the likelihood (which is normally more time-consuming to compute than the prior). Moreover, the likelihood and/or prior (and therefore the posterior) is likely to be vanishingly small at the overwhelming majority of these points (increasingly so in high dimensions), so most of these computations would be pointless anyway.

We could avoid the inefficiency of a grid by sampling from the posterior PDF directly. That is, we would like to be able to draw samples from an arbitrary PDF – let's call it $g(\theta)$ – in such a way that the frequency distribution of the samples is equal to $g(\theta)$ in the limit of a large number of draws. Let's assume that we can evaluate $g(\theta)$ up to some multiplicative constant; that is, we do not require $g(\theta)$ to be normalized. How can we sample from it? This is not self-evident, because in order to sample efficiently we would need to draw preferentially from regions of high relative probability. But it is not obvious how we could locate these regions without explicitly evaluating the function everywhere, which is no better than evaluating on a grid.

It turns out that there are efficient ways to sample an arbitrary probability distribution using *Monte Carlo* methods. We can then use a set of samples $\{\theta\}$ to represent that distribution. This is the case even when $g(\theta)$ is unnormalized: rescaling the distribution by a constant factor will not change the relative frequency with which samples are drawn. Thus once we have such a set of samples we can approximate essentially any property of $g(\theta)$ – such as the mean, variance, and confidence intervals – directly from the set of samples. This applies also to some transformation of the parameter, $f(\theta)$. So if we wanted to calculate the variance of a function f of the samples, we just compute the variance of the transformed samples $\{f(\theta)\}$.

It is also straightforward to use such samples to approximate integrals. Integration is almost endemic in Bayesian inference, so before looking into Monte Carlo sampling methods in section 8.4, let us examine where we need integration (section 8.2) and how we can approximate it using Monte Carlo samples (section 8.3).

8.2 Uses of integration in Bayesian inference

There are at least four different tasks in Bayesian inference that require integration. The first three we have met before.

8.2.1 Marginal parameter distributions

The first task is marginalization, which we already met in section 6.1 and used in section 6.2. If we have a two-dimensional posterior PDF over the parameters (θ_1, θ_2) then we can determine the posterior PDF over just one parameter – that is, regardless of the value of the other parameter – by integration

$$P(\theta_1 \,|\, D, M) \;=\; \int P(\theta_1, \theta_2 \,|\, D, M)\, d\theta_2. \tag{8.3}$$

8.2.2 Expectation values (parameter estimation)

The expectation value of θ is defined as

$$E[\theta] \;=\; \int \theta\, P(\theta \,|\, D, M)\, d\theta. \tag{8.4}$$

This requires $P(\theta \,|\, D, M)$ to be the normalized posterior. If we only have the unnormalized posterior (equation 8.2), then we write this as

$$E[\theta] \;=\; \frac{1}{\int P^*(\theta \,|\, D, M)\, d\theta} \int \theta\, P^*(\theta \,|\, D, M)\, d\theta. \tag{8.5}$$

8.2.3 Model comparison (marginal likelihood)

The denominator in equation 8.1 can be thought of as a normalization constant, because it is the integral of the numerator over all θ

$$P(D \,|\, M) \;=\; \int P(D \,|\, \theta, M) P(\theta \,|\, M)\, d\theta \;=\; \int P^*(\theta \,|\, D, M)\, d\theta. \tag{8.6}$$

But as we shall see in chapter 11, this also plays a central role in model comparison where it is often called the marginal likelihood, or evidence.

8.2.4 Prediction

Suppose we have the data set $D = \{y\}$ obtained at fixed $\{x\}$, and have determined the posterior PDF over the parameters. This model might be a simple straight line, for example. Given a new point x_p, what is the model prediction of y (call it y_p)? The maximum likelihood approach is to find the "best" parameters of the model, to use these in the model equation to predict y, and finally to attempt to propagate the errors in some way (see section 4.1). But this would not take into account the uncertainty in the parameters, which is reflected by the (ignored) finite width of the posterior.

The Bayesian approach is instead to find the posterior PDF over y_p. Specifically, we want to find $P(y_p \,|\, x_p, D, M)$, which is the *posterior predictive distribution* at the specified point x_p given all of the original data D. The model parameters θ do not appear in this expression because in a prediction problem we are not interested in them. The parameters are just a means to an end here. We can remove parameters by marginalizing over them. Dropping M for brevity (it is implicit everywhere), we can therefore write

$$\begin{aligned} P(y_p \,|\, x_p, D) &= \int P(y_p, \theta \,|\, x_p, D)\, d\theta \\ &= \int P(y_p \,|\, x_p, \theta, D)\; P(\theta \,|\, x_p, D)\, d\theta \\ &= \int \underbrace{P(y_p \,|\, x_p, \theta)}_{\text{likelihood}}\; \underbrace{P(\theta \,|\, D)}_{\text{posterior}}\, d\theta. \end{aligned} \tag{8.7}$$

In going from the second line to the third line I have exploited the fact that I am allowed to remove variables to the right of the "|" symbol when the variables to the left are *conditionally independent* of them. $P(y_p \,|\, x_p, \theta, D)$ is independent of D once we specify the model parameters, because for determining the PDF over y_p, D contains no additional information beyond what we have in the parameters. The parameters, together with x_p, are sufficient to determine y_p. In a similar way, the x_p disappears from the other term, $P(\theta \,|\, x_p, D)$, because our knowledge of the model parameters cannot depend on where we later chose to make a prediction. The net result is the third line, which contains two quantities we know: the likelihood and the posterior. This idea of conditional independence is useful when manipulating probability equations because it may lead us to realise that apparently unknown quantities are equivalent to things we already know.

Equation 8.7 tells us that the posterior predictive distribution at point y_p is the posterior-weighted average of the likelihood of y_p. Another way of looking at this is to think that for each θ we get a predictive distribution over y_p (the likelihood). We then average all of these distributions with a weighting factor (the posterior) which tells us how well that θ is supported by the data.

We will see an example of this prediction in action in section 9.1.3.

8.3 Monte Carlo integration

A simple solution to integrating a function is to evaluate it over a dense, regular grid, as we did in chapters 5 and 6. If the N values in this grid are represented by the set $\{\theta_i\}$, then

$$\int f(\theta)\, d\theta \simeq \sum_{i=1}^{N} f(\theta_i)\, \delta\theta = \frac{\Delta\Theta}{N} \sum_{i=1}^{N} f(\theta_i) \tag{8.8}$$

for some function $f(\theta)$, in which $\Delta\Theta$ is the total range of the grid of θ values, and $\delta\theta = \Delta\Theta/N$ is the spacing between them. This method of approximation is called the rectangle method, because we approximate the integral using a sequence of rectangles (figure 3.2). Equation 8.8 is also valid for a irregular grid (e.g. when $\{\theta_i\}$ is drawn from a uniform distribution) in which case $\delta\theta$ is the mean spacing between the grid points.

If θ is a vector of parameters, then the integral in equation 8.8 is multi-dimensional and we think of $\delta\theta$ as being a small N-dimensional hypervolume centered on the sample, and $\Delta\theta$ as being the total hypervolume of the grid. However, as explained in section 8.1, this numerical estimate of the integral suffers from the curse of dimensionality: to keep the uncertainty constant, we would need to increase the number of samples exponentially with the number of dimensions.

We can integrate more efficiently in higher dimensions once we have managed to sample explicitly from a PDF. The Monte Carlo approximation of the integration of some function $f(\theta)$ over the PDF $g(\theta)$ is

$$\frac{\int g(\theta) f(\theta)\, d\theta}{\int g(\theta)\, d\theta} = \langle f(\theta) \rangle \simeq \frac{1}{N} \sum_{i=1}^{N} f(\theta_i) \tag{8.9}$$

where the samples $\{\theta_i\}$ have been drawn from $g(\theta)$, which we assume to be unnormalized. If it were normalized then the denominator on the left would be unity. The symbols $\langle\,\rangle$ denote an expectation value, in this case over $g(\theta)$. The set of samples we get from $g(\theta)$ by Monte Carlo sampling is the same whether or not it is normalized. So if we want to integrate $f(\theta)$ over a normalized PDF $Zg(\theta)$, for some unknown normalization constant Z, then we can safely sample from the unnormalized PDF $g(\theta)$. This is because that constant doesn't make any difference to the relative frequency with which samples are drawn.

Although $g(\theta)$ does not have to be normalized, it does have to be normalizable (proper) so that we can sample from it. The uniform distribution, for example, is only proper if we constrain its range. If $g(\theta) = 1$ is the (unnormalized) uniform distribution over the range $\Delta\Theta$, then $\int g(\theta)\, d\theta = \Delta\Theta$, and equation 8.9 reduces to equation 8.8. So in this particular case the normalization constant of the sampling distribution, $\Delta\Theta$, is still present in the Monte Carlo approximation of the integral.

How precise is the Monte Carlo estimate of the integral? Recalling the discussion in section 2.4, provided the Monte Carlo samples are independent of each other, the uncertainty in $\langle f(\theta) \rangle$ is given by the standard deviation in the mean, which is

$$\begin{aligned} \sigma(\langle f(\theta) \rangle) &= \frac{\sigma_f}{\sqrt{N}} \quad \text{where} \\ \sigma_f &= \sqrt{\frac{1}{N-1} \sum_i \left(f(\theta_i) - \langle f(\theta) \rangle\right)^2} \end{aligned} \tag{8.10}$$

is the sample standard deviation in $f(\theta)$. As σ_f fluctuates around some constant value, the uncertainty in $\langle f(\theta) \rangle$ varies as $N^{-1/2}$: the uncertainty in the Monte Carlo estimate of the integral decreases with the square root of the sample size. This is independent of the dimensionality of the integral, which is precisely the advantage that Monte Carlo methods are intended to bring. This is in contrast to the rectangle method, for which the uncertainty varies as $N^{-1/J}$, where J is the number of dimensions (due to the curse of dimensionality). Thus the Monte Carlo method will generally achieve much smaller errors – or equivalently, will require far fewer function evaluations to achieve a given uncertainty – than grid methods.[1] However, this is only an estimate of the error and not an error bound. The actual performance depends on the specific algorithm.

We can use Monte Carlo integration for all four of the tasks listed in section 8.2.

(1) Suppose we have drawn a set of samples $\{\theta_1, \theta_2\}$ from a two-dimensional distribution $P(\theta_1, \theta_2 \,|\, D, M)$. Each sample is a two-element vector (θ_1, θ_2). Because the set of samples is representative of the full distribution, the set of values $\{\theta_1\}$ is representative of $P(\theta_1 \,|\, D, M)$ (equation 8.3). Thus we marginalize over θ_2 simply by ignoring the values of θ_2.

For the other three cases we use equation 8.9, in which we draw samples from $g(\theta)$ and compute the average of $f(\theta)$ at them.

(2) With $g(\theta)$ equal to the posterior and $f(\theta) = \theta$, we get the *expectation value* of θ (equation 8.4). This tells us that the expectation value of θ with respect to $g(\theta)$ is just the average of samples drawn from $g(\theta)$ (even if it's unnormalized).[2]
(3) With $g(\theta)$ equal to the prior and $f(\theta)$ equal to the likelihood, we get the *marginal likelihood* or evidence (equation 8.6). That is, by drawing samples from the prior and then calculating the likelihood at these, the average of these likelihoods is the evidence. We shall use this in section 11.3.
(4) With $g(\theta)$ equal to the posterior and $f(\theta)$ equal to the likelihood at a new point (x_p, y_p), we obtain the value of the *posterior predictive distribution* at this one point (equation 8.7). That is, we average the likelihood at this new point over the samples drawn from the posterior. We will see in section 9.1.3 how to apply this in practice.

8.4 Monte Carlo sampling

In its most general sense a Monte Carlo method is a means of selecting samples at random to solve what would otherwise be a hard computational problem. Here I use the term to mean sampling from an arbitrary PDF $g(\theta)$, using random numbers drawn from a simpler distribution, i.e. one we can easily draw from.

Having samples drawn *from* $g(\theta)$ is quite different from defining an arbitrary set of values of θ (perhaps a regular grid) at which we then evaluate $g(\theta)$. The power of Monte Carlo sampling is that the samples represent the distribution. Anything we would want to

[1] Note that in one dimension the rectangle method has an error that scales as N^{-1}, so may be more efficient than a Monte Carlo method. The trapezium rule in one dimension even has an error that scales as N^{-2}. But these are only scaling relations. Whether either of these is *actually* more efficient than Monte Carlo depends on the specific problem.

[2] In chapters 5 and 6 we *did* need the normalization constant to calculate the expectation values. That was because we did not have samples *drawn from* the posterior (normalized or not). We instead had evaluations on a regular grid.

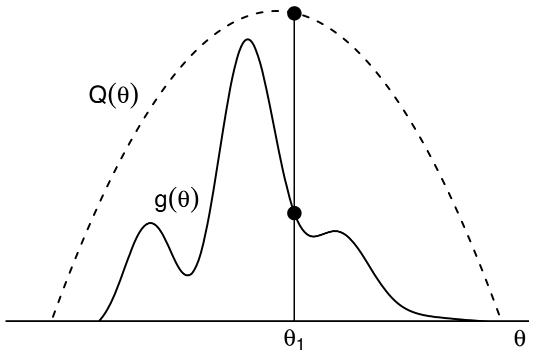

Fig. 8.1 Illustration of rejection sampling of the distribution $g(\theta)$ (solid curved line) using the proposal distribution $Q(\theta)$ (dashed line). We draw a sample θ_1 from $Q(\theta)$. We then draw a random number u from the uniform distribution $\mathcal{U}(0, Q(\theta_1))$. If $u \leq g(\theta_1)$, then θ_1 is accepted to our set of samples. Otherwise it is rejected. We repeat this for a large number of draws from $Q(\theta)$.

do with or compute from the distribution we can approximate using the samples. The mean of the distribution is approximated by the mean of the samples, for example. When we want to plot the distribution, we do not need to know the values of the posterior probability density (normalized or not). We simply take the samples we have drawn from it, and use them to construct a histogram or other density estimate (as done in section 7.2).

We look now at methods of sampling $g(\theta)$. In the following we assume that we can evaluate $g(\theta)$, but we do not require that $g(\theta)$ be normalized: we are free to scale it by any constant.

8.4.1 Rejection sampling

The idea behind rejection sampling is to define a simpler function that we *can* draw samples from, the *proposal distribution*. (We discussed ways to draw random numbers in section 1.8.) Let $Q(\theta)$ be the proposal distribution and define it such that $Q(\theta) \geq g(\theta)$ for all θ. We now draw samples from $Q(\theta)$, but only accept each sample to our set with a probability $g(\theta)/Q(\theta)$; otherwise the sample is rejected. This is illustrated in figure 8.1. The set of samples retained is equivalent to having been drawn from $g(\theta)$.

Unfortunately this method is often inefficient, because in practice we have to use a $Q(\theta)$ that is much larger than $g(\theta)$ at most θ in order for the former to be simple enough to draw from easily. This results in a large number of rejections, so we need a large number of draws and function evaluations. This problem generally gets more acute the higher the dimensionality of $g(\theta)$.

8.4.2 Importance sampling

Importance sampling is a method of calculating expectation values, i.e. the integral in equation 8.9, rather than drawing samples from $g(\theta)$.

Here we again make use of a proposal distribution $Q(\theta)$ that can easily be sampled from (and again this need not be normalized). Having drawn a set of samples $\{\theta_n\}$ from $Q(\theta)$, the idea is to correct for the fact that we have not drawn from $g(\theta)$ by reweighting. Equation 8.9 can be written[3]

$$\langle f(\theta) \rangle = \frac{1}{\int w(\theta)Q(\theta)\,d\theta} \int w(\theta)Q(\theta)f(\theta)\,d\theta \tag{8.11a}$$

$$\simeq \frac{1}{\sum_i w(\theta_i)} \sum_i w(\theta_i)f(\theta_i) \tag{8.11b}$$

where $w(\theta) = g(\theta)/Q(\theta)$ is the *importance weight*, and the sample $\{\theta_i\}$ has now been drawn from $Q(\theta)$. $w(\theta_i)$ is the ratio of the target density to the proposal density at θ_i. This looks like an easy solution, but in order to be easy to draw from, $Q(\theta)$ may be so different from $g(\theta)$ that we still need a very large number of samples.

8.5 Markov Chain Monte Carlo

The main drawback of the above Monte Carlo methods is that they waste a lot of time drawing samples in regions where $g(\theta)$ is low. This happens because we take random draws from the proposal distribution. There is nothing that makes us sample preferentially in regions where $g(\theta)$ is high. We want to do this, but in such a way that the resulting set of samples is still representative of $g(\theta)$. It turns out that we can achieve this if we relax the constraint of drawing samples independently. We cannot draw in an arbitrary fashion, but we can draw using any process that provides samples in the same proportions as $g(\theta)$.

The principle of a *Markov Chain Monte Carlo* (MCMC) method is to set up a random walk over the parameter space which preferentially explores regions of high probability density. The random walk is performed using a *Markov chain*. This is a random process in which the probability of moving from state θ_t to θ_{t+1} is defined by a transition probability $Q(\theta_{t+1}\,|\,\theta_t)$ that depends only on the current state θ_t, and not on any previous ones. This is sometimes referred to as a memoryless process.

We can use this process to select samples from a distribution $g(\theta)$; the sequence of samples is called the *chain*. The key requirement of the Markov chain is that it asymptotically reaches a stationary distribution that is equal to $g(\theta)$. This requirement would not be met if some part of the parameter space over which $g(\theta)$ is defined were not reachable from another part of the parameter space, for example. The requirement can be met if the chain satisfies a condition known as *detailed balance* (which turns out to be a sufficient but not necessary condition). This essentially means that the chain is reversible: if we pick a point

[3] Use equation 8.9 to re-write both the numerator and denominator of equation 8.11a. A term $(1/N)\int Q(\theta)\,d\theta$ cancels to leave equation 8.11b.

and transitioned to another, then it is just as likely that we would pick point θ_a and transition to point θ_b as it is that we would pick point θ_b and transition to point θ_a. Once this has been achieved the chain is in some kind of equilibrium state and the samples we obtain are representative of $g(\theta)$. More information on the required properties of the chains can be found in books on random processes or Monte Carlo methods (e.g. MacKay, 2003). The main point is that not any Markov chain can be used.

There are many different MCMC algorithms including Gibbs sampling, slice sampling, parallel tempering, Hamiltonian Monte Carlo, nested sampling, and the affine-invariant ensemble sampler (Goodman & Weare, 2010; Foreman-Mackey *et al.*, 2013). Here I outline a simple yet widely-used MCMC method, the Metropolis–Hastings algorithm.

8.5.1 Metropolis–Hastings algorithm

As with rejection sampling, the Metropolis–Hastings algorithm also uses a proposal distribution $Q(s \,|\, \theta)$. This is a distribution from which we can easily draw a candidate sample s for the next point in the chain, θ_{t+1}, given the current parameter value θ_t. We initialize the chain at some value. The algorithm then iterates the following two tasks.

(1) Draw at random a candidate sample s from the proposal distribution $Q(s \,|\, \theta_t)$. This distribution could be (and often is) a multivariate Gaussian, in which case the mean is the θ_t, and the covariance matrix specifies the typical size of steps in the chain in each dimension of θ. Generally we want a proposal distribution that gives a higher probability of picking nearby points than far away ones.

(2) Decide whether or not to accept the candidate sample based on the value of the *Metropolis ratio*

$$\rho = \frac{g(s)}{g(\theta_t)} \frac{Q(\theta_t \,|\, s)}{Q(s \,|\, \theta_t)}. \tag{8.12}$$

If $\rho \geq 1$ then we accept the candidate and set $\theta_{t+1} = s$. If $\rho < 1$ we only accept it with a probability of ρ (i.e. we draw a number from a uniform distribution $\mathcal{U}(0,1)$ and compare it to ρ). If we don't accept the candidate then we set $\theta_{t+1} = \theta_t$, i.e. we repeat the existing sample in the chain.

The algorithm iterates between these two tasks and is stopped after some number of iterations. The number required to get a good sampling depends on the problem, but 10^4 to 10^6 iterations is typical.

If we use a symmetric proposal distribution (such as the multivariate Gaussian), then we don't need the term $Q(\theta_t \,|\, s)/Q(s \,|\, \theta_t)$ in the definition of ρ because it is always 1. The algorithm is then often referred to as the Metropolis algorithm (as it's what we had before Hastings came along and generalized it). I provide an R implementation of the Metropolis algorithm using a multivariate Gaussian proposal distribution in section 8.6.2.

Why does the Metropolis algorithm work? The decision rule in task 2 ensures that the walk will always go uphill – towards higher probability densities – if it can. But downhill moves are also allowed. This is essential, otherwise the algorithm will only move toward

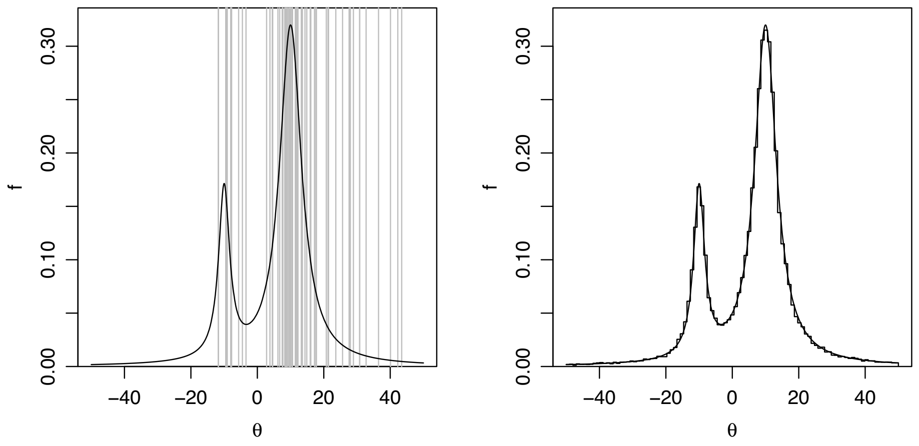

Fig. 8.2 A demonstration of the Metropolis algorithm. The smooth black curve (identical in both panels) is the function we wish to sample from. The left panel overplots 100 individual samples (grey vertical lines). The right panel overplots a histogram built from 10^5 samples.

and then remain at some local maximum, which is not what we want. By also going downhill we can explore the entire distribution. The larger a proposed downhill move, the less likely it is to be accepted. This ensures that regions of higher probability density are preferentially sampled. But given enough iterations even large downhill moves will be made, although we will tend not to stay there, because when near the bottom of a hill most directions lead up. The remarkable thing about this algorithm (not proven here) is that it will, eventually, produce a set of samples that is representative of the entire distribution.

Depending on where in the distribution the chain is initialized, the initial samples may not be representative of it. These should not be retained. The discarded initial samples are called the *burn-in*. With a good initialization the burn-in may only be a few percent of the chain.

A demonstration of the Metropolis algorithm in one dimension is shown in figure 8.2. The function we wish to sample is the smooth black curve. The proposal distribution is a Gaussian with standard deviation $\sigma = 10$. The algorithm is initialized at $\theta = -5$. A burn-in is not used. The left panel shows the first 100 samples (some may be repeated). The right panel show a histogram density estimate of 10^5 samples. R code for running this demonstration in real time, whereby you can see the individual candidate samples being proposed and then accepted or rejected, is provided in section 8.6.1.

8.5.2 Analysing the chains

While the above algorithm works in principle, it may not produce a representative set of samples in practice. This is true for any MCMC method, so we should always inspect the resulting chains to see whether they have the right properties.

There are many ideas concerning what to use as the covariance matrix of the proposal distribution, how long the burn-in period should be, how many iterations we expect to need before convergence, etc. For many proposal distributions the Metropolis algorithm does have the essential property that its stationary distribution is the desired one, and can be reached within a sufficiently large number of iterations. The difficulty in practice is not only that a very large number of samples may be required, but also that there is no simple way of knowing how many samples are sufficient.

One of the simplest ways to check whether the chain has reached some kind of steady state is to rerun the sampling several times, each starting at a different point. All chains should converge to roughly the same region of parameter space. The degree of convergence can be measured by comparing the variance between the chains to the variance within each of the chains. Various metrics exist for quantifying this, such as the one from Gelman & Rubin (1992).

The samples in the chain show some degree of correlation, as the sampling is a Markov chain, not totally random. We can quantify this using the autocorrelation function (ACF). This is the correlation coefficient (equation 1.67) of the chain with an offset version of itself, for different values of this offset, known as the *lag*. Thus the autocorrelation function for a chain of length N at lag h follows from the definition of the sample covariance (equation 1.66) and is

$$\mathrm{ACF}(h) = \frac{\frac{1}{N-h}\sum_{t=1}^{N-h}(\theta_t - \overline{\theta})(\theta_{t+h} - \overline{\theta})}{\frac{1}{N-1}\sum_{t=1}^{N}(\theta_t - \overline{\theta})^2} \tag{8.13}$$

where θ_{t+h} is the chain offset by h steps. $\mathrm{ACF}(h)$ therefore measures how closely the chain is correlated with itself h steps later. It is a rather noisy estimator, so we often need quite long chains for it to be representative. The characteristic length of the chain over which there is a significant correlation is the *autocorrelation length*, defined as

$$\lambda = 1 + 2\sum_{h=1}^{\infty}\mathrm{ACF}(h) \tag{8.14}$$

where in practice the sum has to be truncated to a lower value (at the most N). We can then define the *effective sample size* – the effective number of independent samples in the chain – as N/λ.

Autocorrelation can be a problem because it may make us think the chain has converged to its stationary distribution when in fact it has not. A positive correlation will also lead to an underestimate of the variance of the distribution, if it is computed as the variance of all the samples. This in turn will underestimate the estimated error (equation 8.10 in $\langle f(\theta)\rangle$). The mean and mode will not be affected by the autocorrelations, however.

The degree of correlation we get from MCMC sampling depends on the typical step size, i.e. on the covariance matrix of the proposal distribution. If the steps are large then the chain quickly loses memory of where it has been, so the autocorrelation function decays to small values after a small lag. This suggests we are covering a wide range of the function, which is good, although such large steps may also fail to sample $g(\theta)$ finely enough, which is bad. Conversely, if the steps are small then the ACF drops off slowly because we are sampling a lot around the same place. This is inefficient, and if the chain is too short the resulting PDF will not be representative of $g(\theta)$. We can get some idea of how adventurous the search is by recording the time-variable acceptance rate, the fraction of proposed samples that have been accepted in some time window. This is done by the code in section 8.6.2. We should aim for moderate acceptance rates, say 0.3 to 0.7.

One way to reduce the correlation in a chain after it has been produced is by *thinning*. This means to retain only every Kth sample in the chain ($K > 1$), thereby yielding a less correlated sequence of samples. The drawback of this is that we discard a lot of samples, or rather we must run the algorithm for K times as many iterations to achieve the same number of (retained) samples.

Figure 8.3 plots the evolution of the MCMC chains (left column) and the corresponding autocorrelation function (right column) when sampling the function shown in figure 8.2. The proposal distribution is a Gaussian with standard deviation σ. The top row is for $\sigma = 10$ and no thinning ($K = 1$). The chain has a long correlation length of $\lambda = 203$, and although there are 10^5 samples, there are only 493 effective samples. This nonetheless gives a distribution very close to the true one (right panel of figure 8.2). If we now thin this by a factor of $K = 20$ (the second row), the autocorrelation length is significantly reduced, to 13. The chain plot looks more or less the same, because of the strong correlations when not thinning. Even removing 19 out of every 20 samples does not change the overall features of the chain. A histogram plot using the 5000 remaining samples is slightly noisier than with the full set of 10^5 samples, but looks very similar. Despite the thinning, it turns out that the variance has hardly changed (in this particular case). The bottom two rows show the results for a step size ten times smaller, $\sigma = 1$. Without any thinning the correlation length is large. Thinning significantly improves this, although there is still quite some autocorrelation apparent in the chain.

One big drawback of the Metropolis algorithm is that it uses a fixed step size, the magnitude of which can be hard to determine in advance, yet its value may have a big impact on the efficiency of the sampling. Metropolis is not always the best choice, in particular when there are many dimensions. Kass *et al.* (1998) give some practical advice on using MCMC.

8.5.3 Summarizing the distribution from the sample

We already discussed in section 5.5 the various metrics that can be used to summarize a distribution. Many of these summaries are simple to compute once we have a set of samples $\{\theta\}$ drawn from the distribution. The moments – mean, variance, etc. – can be computed directly from equation 1.42. The quantiles can be computed by first sorting the samples in ascending order, to be $\theta_1, \ldots, \theta_t, \ldots, \theta_N$. The quantile at probability p is approximately $\theta(t = pN)$, whereby we have to think about how to deal with repeated values and how to interpolate. This is done by the R function `quantile`, which has different options for dealing with those issues. If we have multiple parameters, then we could work out the quantiles on the one-dimensional marginalized distributions. The highest density interval (HDI) can be computed from a set of samples using the R function `HPDinterval` in the package `coda`.

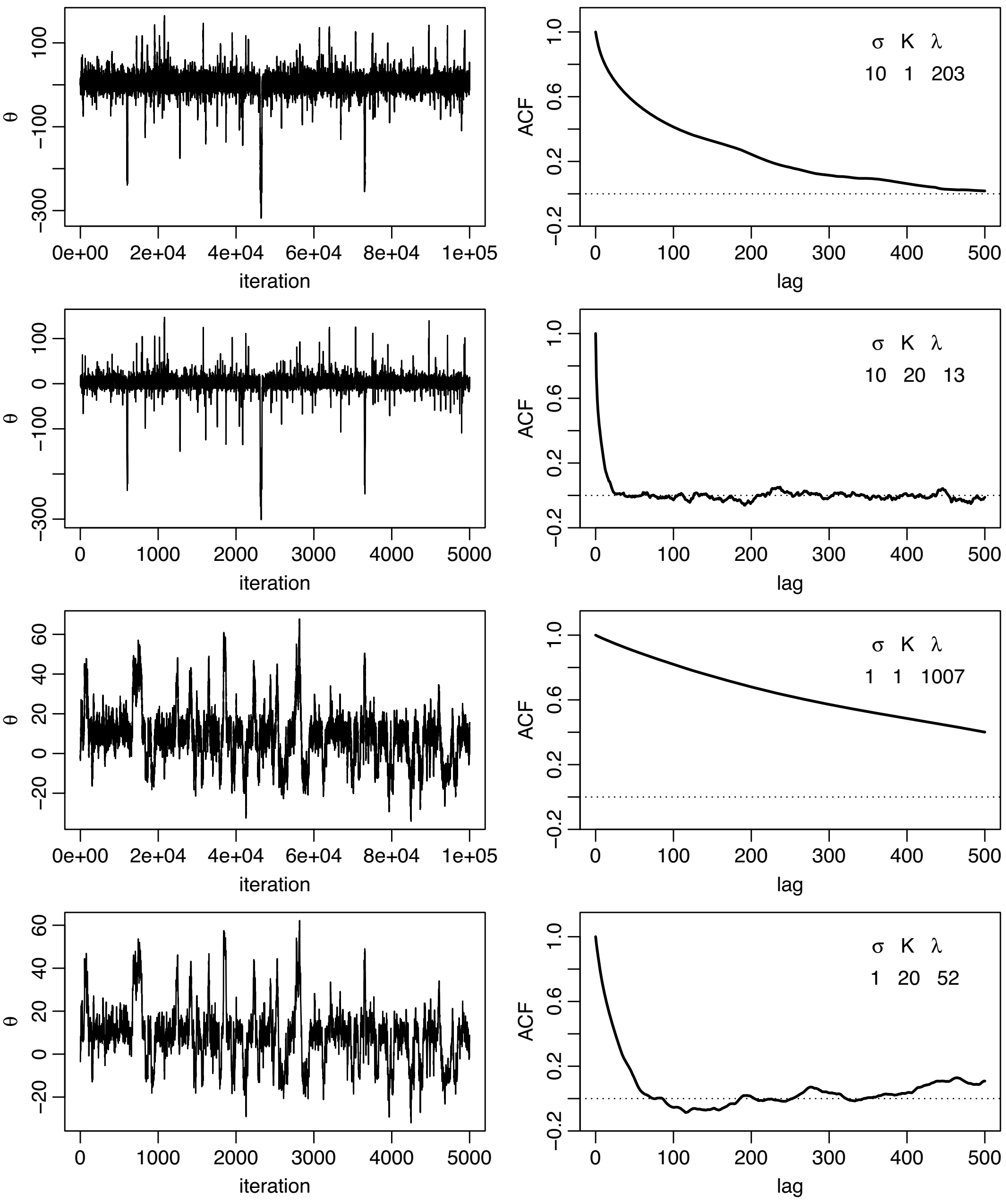

Fig. 8.3 MCMC chains (left column) and the corresponding autocorrelation function (right column) for a Metropolis sampling of the function shown in figure 8.2. In all cases the proposal distribution was a zero mean Gaussian initialized at $\theta = -5$. The four rows correspond to two different values of σ (10 and 1) and two different degrees of the thinning, namely $K = 1$ (no thinning) and $K = 20$. All chains were run for $N = 10^5$ samples before (in the case of $K = 20$) being thinned. λ is the computed correlation length.

8.5.4 Parameter transformations

Sometimes it is more efficient to sample over a transformed parameter. For example, if the parameter θ is strictly positive, then it is more appropriate to sample over (i.e. use a proposal distribution in) $\ln\theta$, as this ensures that proposed values of θ cannot be negative.[4] However, this corresponds to drawing samples from $P(\ln\theta)$, not $P(\theta)$, so this is what we must use in equation 8.12. For this particular example

$$P(\theta)\,d\theta = P(\ln\theta)\,d(\ln\theta) \quad \Rightarrow \quad P(\ln\theta) = \theta P(\theta). \tag{8.15}$$

Thus when we use a symmetric proposal distribution ($Q(\theta_t\,|\,s)/Q(s\,|\,\theta_t) = 1$ in equation 8.12) the Metropolis ratio becomes

$$\rho = \frac{sP(s)}{\theta_t P(\theta_t)}. \tag{8.16}$$

The base of the logarithm in the transformation is unimportant as it corresponds to a constant factor that cancels in the ratio. For a general transformation, the relevant quantity for such transformations is the Jacobian determinant (section 1.9). If we transform from parameters $(\theta_1, \theta_2, \ldots, \theta_J)$ to parameters $(\phi_1, \phi_2, \ldots, \phi_J)$, then we form the Jacobian determinant of the original parameters with respect to the transformed parameters, which is

$$\mathcal{J} = \left| \frac{\partial(\theta_1, \theta_2, \ldots, \theta_J)}{\partial(\phi_1, \phi_2, \ldots, \phi_J)} \right|. \tag{8.17}$$

The Metropolis ratio is then

$$\rho = \frac{P(s)}{P(\theta_t)} \frac{\mathcal{J}_s}{\mathcal{J}_{\theta_t}} \tag{8.18}$$

where the subscripts on the Jacobian determinants indicate the samples they are evaluated at.

8.6 R code

8.6.1 Demonstration of the Metropolis algorithm in one dimension

You can use the following code to sample the test function `testfunc`, or any other function you choose to provide. I recommend you go through the code in chunks rather then sourcing the entire file. With `plotev` set to `TRUE` the code will introduce a delay of `delay` seconds between each iteration. (Actually it's just the plotting that is decelerated; all the samples are calculated in advance by `metrop`.) At each iteration the code plots the current value of the chain in blue, and then the proposed candidate value in red and also prints

[4] We could instead just let the function we want to sample return zero in forbidden regions, as this would prevent such proposals ever being accepted. But this could get quite inefficient if we are near a boundary.

this candidate value to the console. If the proposal is accepted it turns green. If it is not accepted it stays red and the current value turns green, because it is accepted again. The value to be used as the next current value is then printed to the console. The next iteration starts by turning the current point blue. The final plot can either overplot the individual samples or a histogram (density estimate) of the set of samples. The former is only suitable if `Nsamp` is less than a few hundred. I had to fiddle around a bit to get the histogram to appear right (see the comments in the code). I suggest you experiment with different initial values, `thetaInt`, and different step sizes, `sampleSig`.

You can calculate the effective sample size N_{eff} using the function `effectiveSize` in the package `coda`. This actually uses a different formula from that given in section 8.5.2. The syntax is as follows.

```
effectiveSize(as.mcmc(chain$funcSamp[,3]))
```

I then calculate the correlation length as $\lambda = N/N_{\text{eff}}$, where N is the number of samples.

R file: `mcmc_demo.R`

```
##### MCMC demonstration in 1D

plotev    <- TRUE  # do we want to plot the evolution?
delay     <-  1    # time delay in seconds between steps in evolution
thetaInit <- -5    # MCMC initialization
sampleSig <-  10   # MCMC step size
Nsamp     <-  1e5  # number of MCMC samples

source("metropolis.R") # provides metrop()

# Test function
testfunc <- function(theta) {
  return(dcauchy(theta, -10, 2) + 4*dcauchy(theta, 10, 4))
}

# Interface to test function for metrop()
testfunc.metrop <- function(theta) {
  return(c(0,log10(testfunc(theta))))
}

# Establish range, and plot test function
x <- seq(from=-50, to=50, length.out=1e4)
y <- testfunc(x)
ymax <- 1.05*max(y)
# Compute normalization function: used later to put on same as histogram
Zfunc <- sum(y)*diff(range(x))/(length(x))
par(mfrow=c(1,1), mar=c(3.0,3.0,0.5,0.5), oma=c(0.5,0.5,0.5,0.5),
    mgp=c(2.2,0.8,0), cex=1.0)
plot(x, y, type="l", yaxs="i", lwd=2, ylim=c(0, ymax),
     xlab=expression(theta), ylab="f")

# Run MCMC
set.seed(120)
chain <- metrop(func=testfunc.metrop, thetaInit=thetaInit, Nburnin=0,
                Nsamp=Nsamp, sampleCov=sampleSig^2, verbose=Inf, demo=TRUE)
```

```
# Plot evolution of chain to the screen
if(plotev) {
  par(mfrow=c(1,1), mar=c(3.0,3.0,0.5,0.5), oma=c(0.5,0.5,0.5,0.5),
      mgp=c(2.2,0.8,0), cex=1.0)
  for(i in 1:nrow(chain$thetaPropAll)) {
    plot(x, y, type="l", yaxs="i", lwd=2, ylim=c(0, ymax),
         xlab=expression(theta), ylab="f")
    if(i==1) {
      segments(thetaInit, 0, thetaInit, 1, col="blue", lwd=2)
      Sys.sleep(delay)
    } else {
      segments(chain$funcSamp[1:(i-1),3], 0, chain$funcSamp[1:(i-1),3],
               ymax, col="green")
      segments(chain$funcSamp[i-1,3], 0, chain$funcSamp[i-1,3], ymax,
               col="blue", lwd=2)
    }
    lines(x, y, lwd=2) # replot to bring to front
    segments(chain$thetaPropAll[i,1], 0, chain$thetaPropAll[i,1], ymax,
             col="red", lwd=2)
    cat(formatC(i, digits=3), "Proposal: ", formatC(chain$thetaPropAll[i,1],
                                        digits=2, width=6, format="f"))
    Sys.sleep(delay)
    segments(chain$funcSamp[i,3], 0, chain$funcSamp[i,3], ymax,
             col="green", lwd=2)
    cat("  =>  Current: ", formatC(chain$funcSamp[i,3], digits=2, width=6,
                               format="f"), append=TRUE)
    if(chain$funcSamp[i,3]==chain$thetaPropAll[i,1]) {
      cat(" ACCEPTED")
    }
    cat("\n")
    Sys.sleep(delay)
  }
}

# Plot function, samples, and histogram density estimate of samples. The
# histogram is rescaled to have the same normalization as the function.
# Samples outside the range of x are not included in the histogram.
# lines(type="s") plots the vertical steps at the x values given, so we
# must supply it with the histogram breaks, not midpoints. There is one
# more break than counts in the histogram, and each element of hist$counts
# is associated with the lower break, so we must provide an additional
# count=0 to complete the histogram.
pdf("mcmc_demo.pdf", 4, 4)
par(mfrow=c(1,1), mar=c(3.2,3.2,0.5,0.5), oma=c(0.5,0.5,0.5,0.5),
    mgp=c(2.2,0.8,0), cex=1.0)
plot(x, y, type="n", yaxs="i", ylim=c(0, 1.05*max(y)),
     xlab=expression(theta), ylab="f")
# Uncomment the following if you want to plot every sample
#segments(chain$funcSamp[,3], 0, chain$funcSamp[,3], 1, col="green")
lines(x, y, lwd=1) # plot function
# Build and plot histogram of samples
sel <- which(chain$funcSamp[,3]>=min(x) & chain$funcSamp[,3]<=max(x))
hist <- hist(chain$funcSamp[sel,3], breaks=seq(from=min(x), to=max(x),
                                     length.out=100), plot=FALSE)
Zhist <- sum(hist$counts)*diff(range(hist$breaks))/(length(hist$counts))
lines(hist$breaks, c(hist$counts*Zfunc/Zhist,0), type="s", lwd=1)
```

```
dev.off()

# Plot ACF
acor <- acf(chain$funcSamp[,3], lag.max=1e3, plot=FALSE)
plot(acor$lag, acor$acf, xlab="lag", ylab="ACF", type="l", lwd=2)
abline(h=0, lty=3, lwd=1)
```

8.6.2 Implementation of the Metropolis algorithm

The function `metrop` in the file `metropolis.R` is the Metropolis algorithm. It is mostly explained by the inline documentation, but four things should be highlighted. First, it uses a multivariate Gaussian proposal distribution. This is symmetric about its mean, so the factor $Q(\theta_t \,|\, s)/Q(s \,|\, \theta_t)$ is always 1 in the definition of the Metropolis ratio (equation 8.12). Second, the algorithm works with the logarithm of the Metropolis ratio. This is because it also uses the logarithm of the density of the function being sampled (for dynamic range reasons), so we save computation time and retain this dynamic range if we also make the selection in the logarithm. Third, the R function that `metrop` calls, `func`, must return a two-element vector, the sum of which is the logarithm of the density of the function being sampled. I designed it this way because in Bayesian applications I use `metrop` to sample an unnormalized posterior, which is the product of a prior and a likelihood. I then define `func` to return the logarithm of the prior and the logarithm of the likelihood, the sum of which is the logarithm of the unnormalized posterior. It is useful to have these available separately for the sake of subsequent analyses.[5] It is also useful when using other techniques that use the individual likelihoods. One example is the cross-validation likelihood technique, to be discussed in section 11.6.1. If we didn't return the likelihood values we would have to recalculate them. If you want to use `metrop` to sample a distribution that does not separate into the product of a prior and a likelihood, just define `func` so that one of the two elements it returns is zero. This is what I did in the function `testfunc.metrop` in the demonstration code listed in the previous section. Finally, the initial values of the parameters, `thetaInit`, passed into `metrop` must produce finites value of the function, and not zero, `NA`, `NaN`, etc., otherwise the code will throw an error. Ensure you use a sensible initialization.

R file: `metropolis.R`

```
##### The Metropolis algorithm

library(mvtnorm) # for rmvnorm

# Metropolis (MCMC) algorithm to sample from function func.
# The first argument of func must be a real vector of parameters,
# the initial values of which are provided by the real vector thetaInit.
# func() returns a two-element vector, the logPrior and logLike
# (log base 10), the sum of which is taken to be the log of the density
# function (i.e. unnormalized posterior). If you don't have this separation,
# just set func to return one of them as zero. The MCMC sampling PDF is the
# multivariate Gaussian with fixed covariance, sampleCov. A total of
# Nburnin+Nsamp samples are drawn, of which the last Nsamp are kept. As the
```

[5] Some implementations of MCMC algorithms don't return the function values at all.

```
# sampling PDF is symmetric, the Hasting factor cancels, leaving the basic
# Metropolis algorithm. Diagnostics are printed very verbose^th sample:
# sample number, acceptance rate so far.
# ... is used to pass data, prior parameters etc. to func().
# If demo=FALSE (default), then
# return a Nsamp * (2+Ntheta) matrix (no names), where the columns are
# 1:  log10 prior PDF
# 2:  log10 likelihood
# 3+: Ntheta parameters
# (The order of the parameters in thetaInit and sampleCov must match.)
# If demo=TRUE, return the above (funcSamp) as well as thetaPropAll, a
# Nsamp * Ntheta matrix of proposed steps, as a two element named list.
metrop <- function(func, thetaInit, Nburnin, Nsamp, sampleCov, verbose,
                   demo=FALSE, ...) {

  Ntheta   <- length(thetaInit)
  thetaCur <- thetaInit
  funcCur  <- func(thetaInit, ...) # log10
  funcSamp <- matrix(data=NA, nrow=Nsamp, ncol=2+Ntheta)
  # funcSamp will be filled and returned
  nAccept  <- 0
  acceptRate <- 0
  if(demo) {
    thetaPropAll <- matrix(data=NA, nrow=Nsamp, ncol=Ntheta)
  }

  for(n in 1:(Nburnin+Nsamp)) {

    # Metropolis algorithm. No Hastings factor for symmetric proposal
    if(is.null(dim(sampleCov))) { # theta and sampleCov are scalars
      thetaProp <- rnorm(n=1, mean=thetaCur, sd=sqrt(sampleCov))
    } else {
      thetaProp <- rmvnorm(n=1, mean=thetaCur, sigma=sampleCov,
                           method="eigen")
    }
    funcProp  <- func(thetaProp, ...)
    logMR <- sum(funcProp) - sum(funcCur) # log10 of the Metropolis ratio
    #cat(n, thetaCur, funcCur, ":", thetaProp, funcProp, "\n")
    if(logMR>=0 || logMR>log10(runif(1, min=0, max=1))) {
      thetaCur   <- thetaProp
      funcCur    <- funcProp
      nAccept    <- nAccept + 1
      acceptRate <- nAccept/n
    }
    if(n>Nburnin) {
      funcSamp[n-Nburnin,1:2] <- funcCur
      funcSamp[n-Nburnin,3:(2+Ntheta)] <- thetaCur
      if(demo) {
        thetaPropAll[n-Nburnin,1:Ntheta] <- thetaProp
      }
    }

    # Diagnostics
    if( is.finite(verbose) && (n%%verbose==0 || n==Nburnin+Nsamp) ) {
      s1 <- noquote(formatC(n,          format="d", digits=5, flag=""))
      s2 <- noquote(formatC(Nburnin,    format="g", digits=5, flag=""))
      s3 <- noquote(formatC(Nsamp,      format="g", digits=5, flag=""))
      s4 <- noquote(formatC(acceptRate, format="f", digits=4, width=7,
                            flag=""))
      cat(s1, "of", s2, "+", s3, s4, "\n")
    }

  }

  if(demo) {
    return(list(funcSamp=funcSamp, thetaPropAll=thetaPropAll))
  } else {
    return(funcSamp)
  }

}
```