Some Notes on Markov Chain Monte Carlo (MCMC)

John Fox & Georges Monette

2016-12-30

1 Introduction

These notes are meant to describe, explain (in a non-technical manner), and illustrate the use of Markov Chain Monte Carlo (MCMC) methods for sampling from a distribution. Section 2 takes up the original MCMC method, the Metropolis-Hastings algorithm, outlining the algorithm and applying it to sampling from a bivariate-normal distribution. Section 3 is similar, but describes the Gibbs sampler. Section 4 explains Hamiltonian Monte Carlo, the current state-of-the-art MCMC method. Section 5 applies the Metropolis-Hastings algorithm to a simple one-parameter problem in Bayesian inference, sampling from the posterior distribution of a proportion.

In compiling this document we found the following sources particularly helpful: Chib and Greenberg (1995), for the Metropolis-Hastings algorithm; Casella and George (1992), for the Gibbs sampler; and Neal (2011) for Hamiltonian MCMC. An excellent general source on MCMC methods is Gelman et al. (2013).

2 The Metropolis-Hastings Algorithm

What's come to be called the Metropolis-Hastings algorithm was originally formulated by Metropolis et al. (1953) and subsequently generalized by Hastings (1970). We'll explain the more general version of the algorithm, but will use the orginal, simpler version in an example.

Problem: We have a continuous vector random variable $\mathbf{x}$ with n elements and with density function $p(\mathbf{x})$. We don't know how to compute $p(\mathbf{x})$ but we do have a function proportional to it, $p^*(\mathbf{x}) = c \times p(\mathbf{x})$, where $c = \int_{\mathbf{x}} p^*(\mathbf{x}) d\mathbf{x}$. (We don't know c or we'd know $p(\mathbf{x})$.) We want to draw random samples from the *target* distribution $p(\mathbf{x})$. One way this situation might arise is in Bayesian inference, where $p^*(\cdot)$ could be the unnormalized posterior, computed as the product of the prior density and the likelihood (see the example in Section 5).

The Metropolis-Hastings algorithm starts with an arbitrary value $\mathbf{x}_0$ of $\mathbf{x}$, and proceeds to generate a sequence of m realized values $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_{i-1}, \mathbf{x}_i, \ldots, \mathbf{x}_m$. Each subsequent realized value is sampled randomly based on a *candidate* or *proposal* distribution, with density function $f(\mathbf{x}_i|\mathbf{x}_{i-1})$, from which we know how to sample. As the notation implies, the proposal distribution employed depends only on the immediately preceding value of $\mathbf{x}$. The next value of $\mathbf{x}$ sampled from the proposal distribution may be accepted or rejected,

hence the term "proposal" or "candidate." If the proposed value of $\mathbf{x}_i$ is rejected, then the preceding value is retained; that is, $\mathbf{x}_i$ is set to $\mathbf{x}_{i-1}$. This procedure defines a Markov process yielding a Markov chain of values, because the probability of transition from one state to another (one value of $\mathbf{x}$ to the next) depends only on the previous state.

Within broad regularity conditions, the choice of proposal distribution is arbitrary. For example, it's necessary that the proposal distribution and initial value $\mathbf{x}_0$ lead to a Markov process capable of visiting the complete support of $\mathbf{x}$—that is, all values of $\mathbf{x}$ for which the density $p(\mathbf{x})$ in nonzero. And different choices of proposal distributions may be differentially desirable, for example, in the sense that they are more or less efficient—tend to require fewer or more generated values to cover the support of $\mathbf{x}$ thoroughly.

With this background, the Metropolis-Hastings algorithm proceeds as follows. For each $i = 1, 2, \ldots, m$:

1. Sample a candidate value $\mathbf{x}^*$ from the proposal distribution $f(\mathbf{x}_i|\mathbf{x}_{i-1})$.

2. Compute the *acceptance ratio*

$$\begin{aligned} a &= \frac{p(\mathbf{x}^*)f(\mathbf{x}^*|\mathbf{x}_{i-1})}{p(\mathbf{x}_{i-1})f(\mathbf{x}_{i-1}|\mathbf{x}^*)} \\ &= \frac{p^*(\mathbf{x}^*)f(\mathbf{x}^*|\mathbf{x}_{i-1})}{p^*(\mathbf{x}_{i-1})f(\mathbf{x}_{i-1}|\mathbf{x}^*)} \end{aligned} \tag{1}$$

   Notice that the substitution of $p^*(\cdot)$ for $p(\cdot)$ in the second line of Equation 1 is justified because the unknown normalizing constant c cancels in the numerator and denominator, making the ratio in the equation computable. Compute $a' = \min(a, 1)$.

3. Generate a uniform random number u on the unit interval, $U \sim \text{unif}(0, 1)$. If $u \leq a'$, set the ith value in the chain to the proposal, $\mathbf{x}_i = \mathbf{x}^*$; otherwise retain the previous value, $\mathbf{x}_i = \mathbf{x}_{i-1}$. In effect, the proposal is accepted with certainty if it is "at least as probable" as the preceding value, taking into account the possible bias in the direction of movement of the proposal function from the preceding value. If the proposal is less probable than the preceding value, then the probability of accepting the proposal declines with the ratio a, but isn't 0. Thus, the chain will tend to visit higher-density regions of the target distribution with greater frequency but will tend to explore the entire target distribution. It can be shown (e.g., Chib and Greenberg, 1995) that the limiting distribution of the Markov chain is indeed the target distribution, and so the algorithm should work if m is big enough.

The Metropolis-Hastings algorithm is simpler when the proposal distribution is symmetric, in the sense that $f(\mathbf{x}_i|\mathbf{x}_{i-1}) = f(\mathbf{x}_{i-1}|\mathbf{x}_i)$. This is true, for example, when the proposal distribution is multivariate-normal with mean vector $\mathbf{x}_{i-1}$ and some specified covariance matrix $\mathbf{S}$:

$$\begin{aligned} f(\mathbf{x}_i|\mathbf{x}_{i-1}) &= \frac{1}{(2\pi)^{n/2}\sqrt{\det \mathbf{S}}} \times \exp\left[-\frac{1}{2}(\mathbf{x}_i - \mathbf{x}_{i-1})'\mathbf{S}^{-1}(\mathbf{x}_i - \mathbf{x}_{i-1})\right] \\ &= f(\mathbf{x}_{i-1}|\mathbf{x}_i) \end{aligned} \tag{2}$$

Then, a in Equation 1 becomes

$$a = \frac{p^*(\mathbf{x}^*)}{p^*(\mathbf{x}_{i-1})} \tag{3}$$

which (again, because the missing normalizing constant c cancels) is equivalent to the ratio of the target density at the proposed and preceding values of $\mathbf{x}$. This simplified version of the Metropolis-Hastings algorithm, based on a symmetric proposal distribution, is the version given originally by Hastings (1970).

By construction, the Metropolis-Hastings algorithm generates statistically *dependent* successive values of $\mathbf{x}$. If an approximately independent sample is desired, the sequence of sampled values can be *thinned* by discarding a sufficient number of intermediate values of $\mathbf{x}$, retaining only every kth value. Likewise, because of an unfortunately selected initial value $\mathbf{x}_0$, it may take some time for the sampled sequence to approach its limiting distribution—that is, the target distribution. Consequently, it may be advantageous to discard a number of values at the beginning of the sequence, termed the *burn-in period*.

2.1 Using the Metropolis Algorithm to Sample From a Bivariate-Normal Distribution

In this section, we'll demonstrate the Metropolis algorithm by drawing samples from a bivariate-normal distribution with the following (arbitrary) mean vector and covariance matrix:

$$\begin{aligned} \boldsymbol{\mu} &= \begin{pmatrix} 1 \\ 2 \end{pmatrix} \\ \boldsymbol{\Sigma} &= \begin{pmatrix} 1 & 1 \\ 1 & 4 \end{pmatrix} \end{aligned} \tag{4}$$

It's not necessary, of course, to use MCMC in this case, because it's simple to draw multivariate-normal samples directly, but the bivariate-normal distribution provides a simple setting in which to demonstrate the algorithm, and it helps to know the right answer in advance. We'll pretend that we know the bivariate-normal distribution only up to a constant of proportionality. To this end, we omit the normalizing constant, which for this simple example works out to $2\pi \times \sqrt{3}$).

```
> # packages required:
> library(mvtnorm)
> library(MASS)
> library(ks)

Loading required package:  KernSmooth
KernSmooth 2.23 loaded
Copyright M. P. Wand 1997-2009
Loading required package:  misc3d
Loading required package:  rgl
```

```
> library(car)
> library(rgl)  # for 3D graphics
>
>
> # density up to a constant of proportionality
>
> p.star <- function(x, mu, Sigma){
+     exp(- 0.5 * (x - mu) %*% solve(Sigma) %*% (x - mu))
+ }
>
> mu <- c(1, 2)
> Sigma <- c(1, 1, 1, 4)
> Sigma <- matrix(Sigma, 2, 2)
>
> # check
>
> some.xs <- list(c(1, 2), c(0, 0), c(3, 3))
> sapply(some.xs, p.star, mu=mu, Sigma=Sigma) /
+     sapply(some.xs, dmvnorm, mean=mu, sigma=Sigma)

[1] 10.8828 10.8828 10.8828

> 2*pi*sqrt(3)  # check normalizing constant

[1] 10.8828
```

We'll use a bivariate-rectangular proposal distribution centered at the preceding value $\mathbf{x}_{i-1}$ with half-extent $\delta_1 = 2$ in the direction of the coordinate x_1 and $\delta_2 = 4$ in the direction of x_2. This distribution is symmetric, as required by the simpler Metropolis algorithm. Clearly, because it has finite support, the proposal distribution doesn't cover the entire support of the bivariate-normal distribution, but because the proposal distribution travels with $\mathbf{x}_i$, it can generate a valid Markov chain. We arbitrarily set $\mathbf{x}_0 = (0, 0)'$, and sample $m = 10^5$ values of $\mathbf{x}$, keeping track of how many proposals are accepted and how many are rejected:

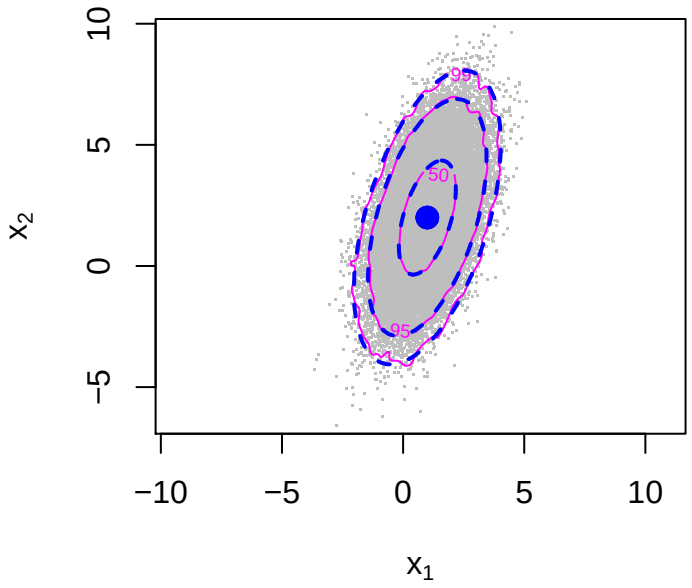
```
> m <- 1e5  # draws
> x.current <- c(0, 0)  # x_0
> xs <- matrix(0, m, 2)  # to hold sampled values
>
> set.seed(811018)  # for reproducibility
>
> delta <- c(2, 4)  # for bivariate uniform proposal distribution
>
> rbvunif <- function(mu, delta){
+     u1 <- runif(1, mu[1] - delta[1], mu[1] + delta[1])
```

```
+     u2 <- runif(1, mu[2] - delta[2], mu[2] + delta[2])
+     c(u1, u2)
+ }
>
> accepted <- rejected <- 0
>
> system.time(
+     for (i in 1:m){
+         x.proposed <- rbvunif(mu=x.current, delta=delta)  # proposal
+         a <- p.star(x.proposed, mu, Sigma) /
+             p.star(x.current, mu, Sigma)  # aceptance ratio
+         if (a >= 1 || runif(1) <= a) {
+             xs[i, ] <- x.proposed
+             x.current <- x.proposed
+             accepted <- accepted + 1
+         }
+         else {
+             xs[i, ] <- x.current
+             rejected <- rejected + 1
+         }
+     }
+ )

   user  system elapsed
   7.58    0.00    7.57

> accepted/m

[1] 0.41678

> accepted + rejected

[1] 1e+05
```

How well did the Metropolis algorithm do in approximating the bivariate-normal density? We'll examine the means and covariance matrix for the sampled values. We'll also compare empirical density contours of the sampled values, produced by a nonparametric bivariate density estimate (using the `kde()` function in the **ks** package), with the corresponding theoretical elliptical contours of the bivariate-normal distribution (drawn with the `ellipse()` function in the **car** package). We use the `eqscplot()` function from the **MASS** package to insure that the graph is drawn with equal units/cm for the axes:

```
> # checks:
>
> colMeans(xs)
```

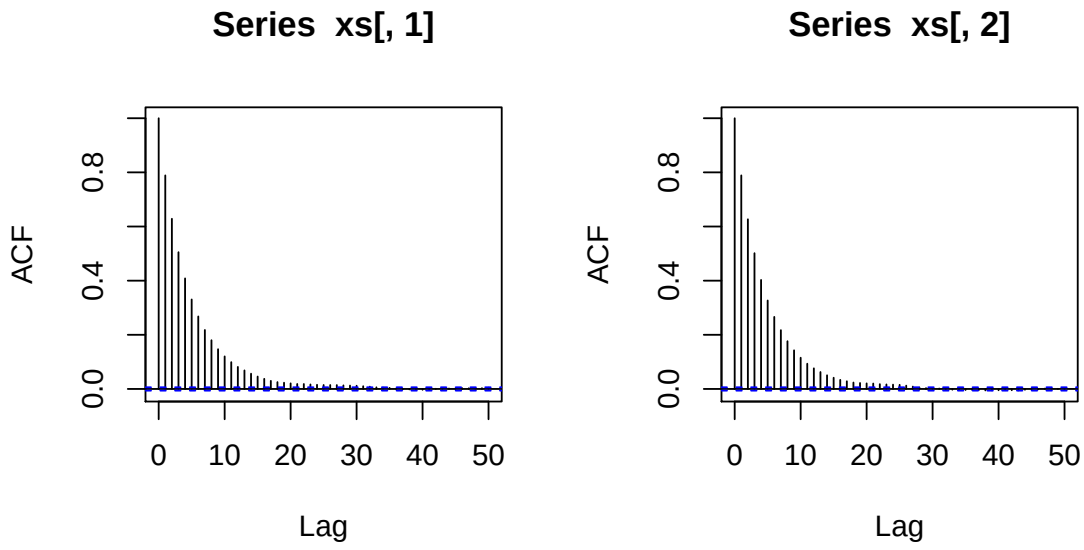
```
[1] 1.002813 1.986911

> var(xs)

          [,1]      [,2]
[1,] 0.9891840 0.9721024
[2,] 0.9721024 3.9627934

> eqscplot(xs, pch=".", col="gray", xlab=expression(x[1]),
+          ylab=expression(x[2]))
> res <- kde(xs)
> plot(res, add=TRUE, cont=c(50, 95, 99), col="magenta")
> ellipse(mu, Sigma, col='blue', radius=sqrt(qchisq(.5, 2)), lty=2)
> ellipse(mu, Sigma, col='blue', radius=sqrt(qchisq(.95, 2)), lty=2)
> ellipse(mu, Sigma, col='blue', radius=sqrt(qchisq(.99, 2)), lty=2)
```



Clearly, the sampled values do a good job of reproducing the target distribution.

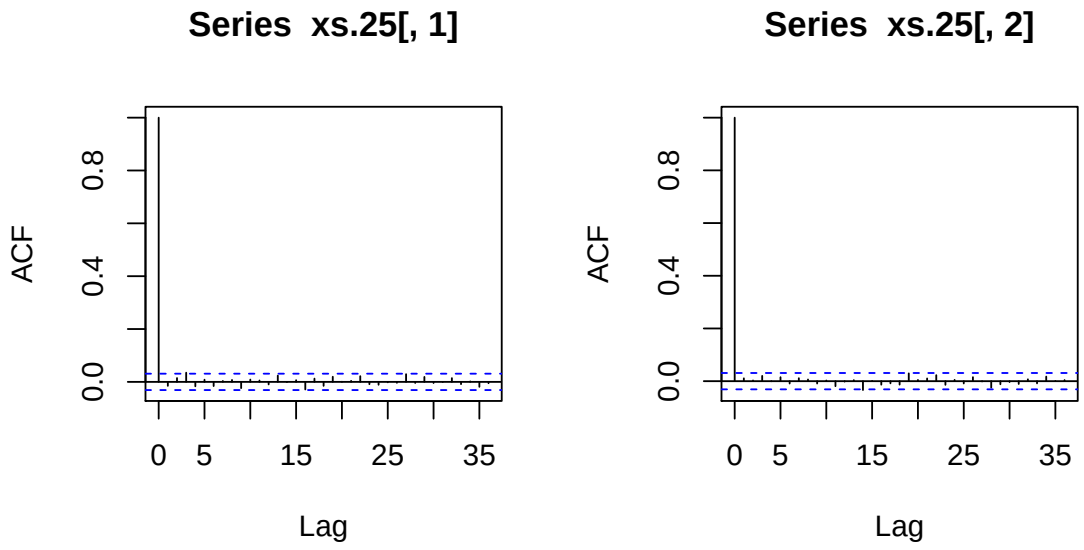
Next, we'll check the autocorrelation functions of the sampled values:

```
> par(mfrow=c(1, 2))
> acf(xs[, 1])
> acf(xs[, 2])
```



The autocorrelations are large at small lags, but decay to near 0 by lag 25, which suggests thining by selecting each 25th value to produce an approximately independent sample:

```
> # thinning
> xs.25 <- xs[seq(25, 1e5, by=25), ]
> par(mfrow=c(1, 2))
> acf(xs.25[, 1])
> acf(xs.25[, 2])
```

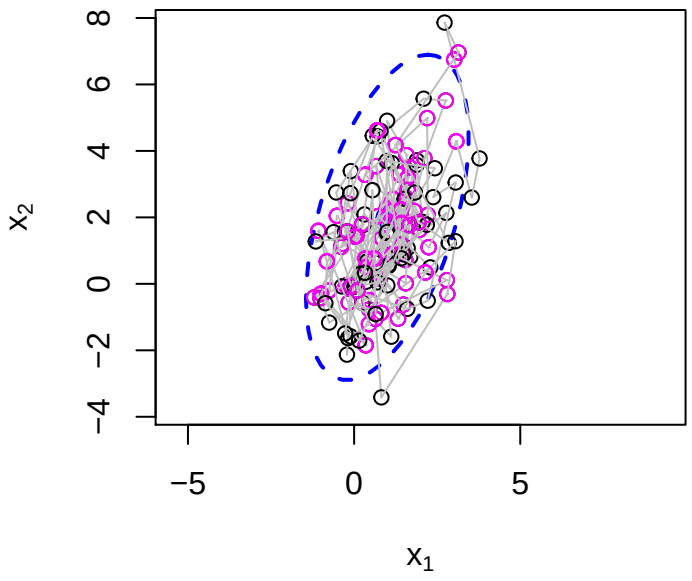


As expected, the autocorrelations now are all small.

Finally, here's some R code that (when run interactively—only the final result is shown

here) produces an animation of the first 300 sampled points, with successive points connected by line-segments. The theoretical 95% concentration ellipse is superimposed on the graph; the magenta points represent rejected proposals (and hence repeated successive values), while the black points represent accepted proposals:

```
> # animation
>
> eqscplot(0, 0, type="n", xlim=c(-4, 8), ylim=c(-4, 8),
+          xlab=expression(x[1]), ylab=expression(x[2]))
> ellipse(mu, Sigma, col='blue', radius=sqrt(qchisq(.95, 2)), lty=2)
>
> m <- 300
> pb <- txtProgressBar(1, m, initial=1, style=3)
> points(xs[1, 1], xs[1, 2])
> for (i in 2:m){
+     Sys.sleep(0.1)
+     setTxtProgressBar(pb, i)
+     lines(xs[c(i, i - 1), ], col="gray")
+     points(xs[i, 1], xs[i, 2],
+            col = if(all(xs[i, ] == xs[i - 1,])) "magenta" else "black")
+ }
> close(pb)
```



3 The Gibbs Sampler

The Gibbs sampler is an MCMC algorithm originally developed for applications in image processing by Geman and German (1984), who named it after the 19th Century physicist Josiah Gibbs. Gelfand and Smith (1990) pointed out the applicability of the Gibbs sampler to statistical problems.

The simple Gibbs sampler, described below, is based on the observation that the joint distribution of an n-dimensional vector random variable $\mathbf{x}$ can be composed from the conditional distribution of each of its elements given the others, that is $p(X_j|\mathbf{x}_{-j})$ for $j = 1, 2, \ldots, n$ (where $\mathbf{x}_{-j} = [X_1, X_2, \ldots, X_{j-1}, X_{j+1}, \ldots, X_n]'$ is $\mathbf{x}$ with the jth element removed). Although it was developed independently, in this basic form, the Gibbs sampler turns out to be a special case of the general Metropolis-Hastings algorithm (see Gelman et al., 2013, p. 281).

There are many variations on the Gibbs sampler, such as its application to subsets of $\mathbf{x}$ (some of which are of size greater than 1) that partition $\mathbf{x}$: That is, with suitable ordering of its elements, $\mathbf{x} = [\mathbf{x_1}', \mathbf{x_2}', \ldots, \mathbf{x_q}']'$. The coresponding conditional distributions are $p(\mathbf{x_j}|\mathbf{x_{-j}})$. Conditional distributions of this form can arise naturally in the process of specifying Bayesian statistical models in circumstances where it is difficult to derive the joint distribution $p(\mathbf{x})$ analytically.

The basic Gibbs sampler is simple to describe and proceeds as follows:

1. Pick an arbitrary set of initial values $\mathbf{x} = \mathbf{x}_0$.

2. Then for each of m iterations, sample in succession each element of $\mathbf{x}$ from its conditional distribution, conditioning on the most recent values of the other elements. That is for $i = 1, 2, \ldots, m$:

   Sample $x_1^{(i)}$ from $p(x_1|X_2 = x_2^{(i-1)}, \ldots, X_n = x_n^{(i-1)})$.
   Sample $x_2^{(i)}$ from $p(x_2|X_1 = x_1^{(i)}, X_3 = x_3^{(i-1)}, \ldots, X_n = x_n^{(i-1)})$.
   $\vdots$
   Sample $x_n^{(i)}$ from $p(x_n|X_1 = x_1^{(i)}, \ldots, X_{n-1} = x_{n-1}^{(i)})$.
   Save $\mathbf{x}_i = [x_1^{(i)}, x_2^{(i)}, \ldots, x_n^{(i)}]'$.

3.1 Using the Gibbs Sampler to Sample From a Bivariate-Normal Distribution

As we did for the Metropolis algorithm in Section 2.1, we'll illustrate the Gibbs sampler by drawing samples from a bivariate-normal distribution with mean vector and covariance matrix

$$\begin{aligned} \boldsymbol{\mu} &= \begin{pmatrix} 1 \\ 2 \end{pmatrix} \\ \boldsymbol{\Sigma} &= \begin{pmatrix} 1 & 1 \\ 1 & 4 \end{pmatrix} \end{aligned} \tag{5}$$

As before, this example is artificial because (as previously mentioned) it's easy to sample directly from the bivariate-normal distribution.

To apply the Gibbs sampler, we need the conditional distributions $p(x_1|x_2)$ and $p(x_2|x_1)$. In the bivariate-normal case, the conditional distributions are normal and are provided by the population linear regressions of each variable on the other. That is, the regression of X_1 on X_2 is

$$E(X_1|x_2) = \alpha_{12} + \beta_{12}x_2 \tag{6}$$

where

$$\begin{aligned} \beta_{12} &= \frac{\sigma_{12}}{\sigma_2^2} \\ \alpha_{12} &= \mu_1 - \beta_{12}\mu_2 \end{aligned} \tag{7}$$

with (constant) error varianace

$$\sigma^2_{1|2} = \sigma_1^2\left(1 - \frac{\sigma_{12}^2}{\sigma_1^2\sigma_2^2}\right) \tag{8}$$

In these equations, μ_1 and μ_2 are the unconditonal means of X_1 and X_2, σ_1^2 and σ_2^2 are their variances, and σ_{12} is their covariance. Thus

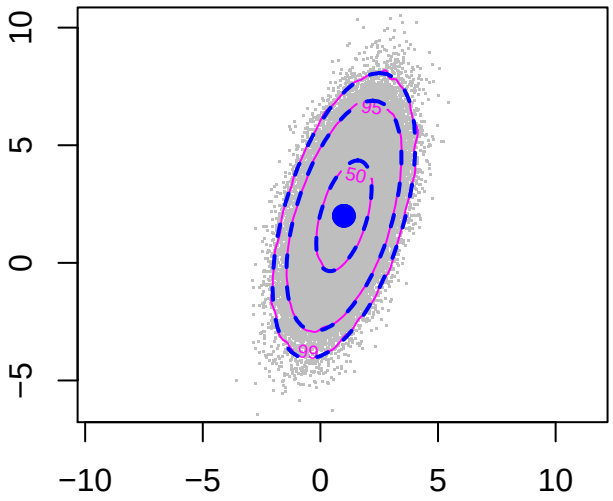
$$X_1|x_2 \sim N\left(\alpha_{12} + \beta_{12}x_2, \sigma^2_{1|2}\right) \tag{9}$$

The results for the conditional distribution of X_2 given $X_1 = x_1$ are entirely analogous.

Programming the Gibbs sampler for this example in R is straightforward. In doing so, we'll follow the outline of the similar example for the Metropolis algorithm that we developed in Section 2.1:

```
> mu # from before

[1] 1 2

> Sigma # from before

     [,1] [,2]
[1,]    1    1
[2,]    1    4

> m <- 1e5  # draws
> x.current <- c(0, 0)  # x_0
> xs <- matrix(0, m, 2)  # to hold sampled values
>
> set.seed(900323)  # for reproducibility
>
> # regressions:
>
> beta_1.2  <- Sigma[1, 2]/Sigma[2, 2]
```

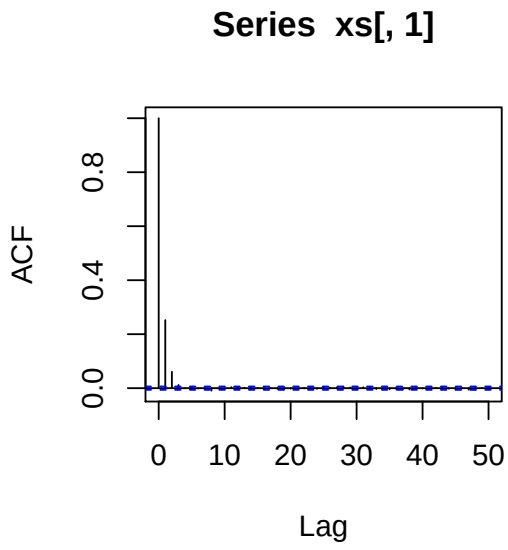
```
> alpha_1.2 <- mu[1] - beta_1.2*mu[2]
> beta_2.1  <- Sigma[2, 1]/Sigma[1, 1]
> alpha_2.1 <- mu[2] - beta_2.1*mu[1]
> sigma_1.2 <- sqrt(Sigma[1, 1]*(1 - Sigma[1, 2]^2/
+                     (Sigma[1, 1]*Sigma[2, 2])))
> sigma_2.1 <- sqrt(Sigma[2, 2]*(1 - Sigma[2, 1]^2/
+                     (Sigma[1, 1]*Sigma[2, 2])))
>
> # Gibbs iterations
> for (i in 1:m){
+     x.current[1] <- rnorm(1, mean = alpha_1.2 + beta_1.2*x.current[2],
+                           sd = sigma_1.2)
+     x.current[2] <- rnorm(1, mean = alpha_2.1 + beta_2.1*x.current[1],
+                           sd = sigma_2.1)
+     xs[i, ] <- x.current
+ }
>
> # checks:
>
> colMeans(xs)

[1] 1.007277 2.016553

> var(xs)

          [,1]     [,2]
[1,] 0.9965732 1.002036
[2,] 1.0020355 4.021808

> eqscplot(xs, pch=".", col="gray")
> res <- kde(xs)
> plot(res, add=TRUE, cont=c(50, 95, 99), col="magenta")
> ellipse(mu, Sigma, col='blue', radius=sqrt(qchisq(.5, 2)), lty=2)
> ellipse(mu, Sigma, col='blue', radius=sqrt(qchisq(.95, 2)), lty=2)
> ellipse(mu, Sigma, col='blue', radius=sqrt(qchisq(.99, 2)), lty=2)
```

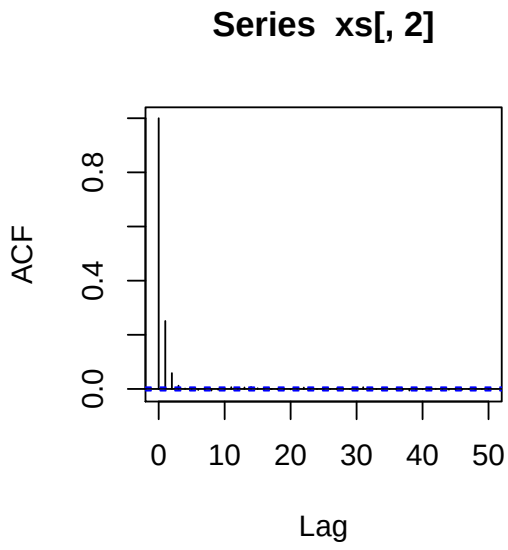


Evidently, the Gibbs sampler accurately recovers the means, variances, covariance, and shape of the bivariate-normal distribution.

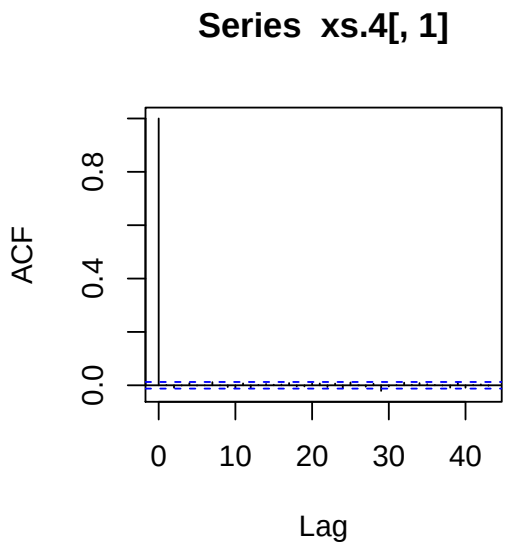
The sampled values of X_1 and X_2 are autocorrelated, but less so than those produced for this example by the Metropolis algorithm. Consquently, we can get an approximately independent sample with less thinning (taking every 4th value):

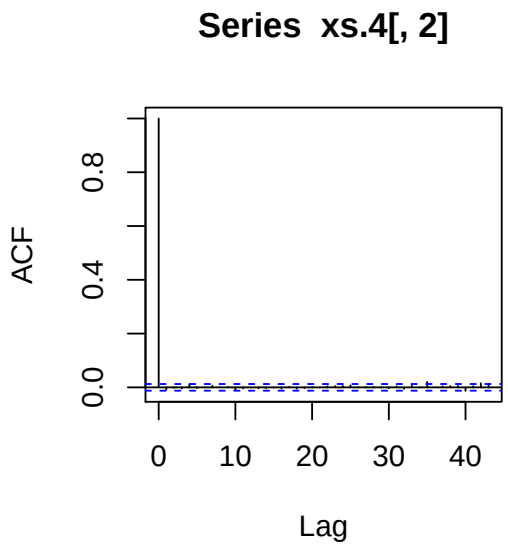
```
> par(mfrow=c(1, 2))
> acf(xs[, 1])
> acf(xs[, 2])
```





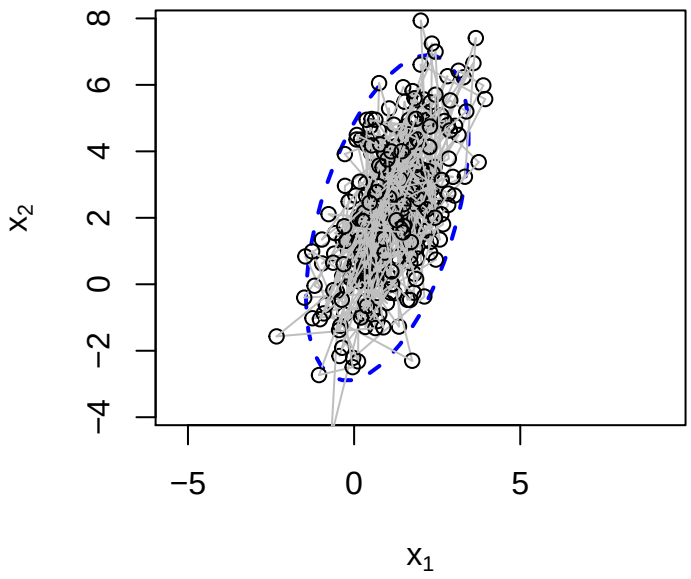

```
> # thinning
>
> xs.4 <- xs[seq(4, 1e5, by=4), ]
> acf(xs.4[, 1])
> acf(xs.4[, 2])
```





As for the Metropolis algorithm, here's an interactive animation of Gibbs sampling (with only the final result shown):

```
> # animation
>
> eqscplot(0, 0, type="n", xlim=c(-4, 8), ylim=c(-4, 8),
+          xlab=expression(x[1]), ylab=expression(x[2]))
> ellipse(mu, Sigma, col='blue', radius=sqrt(qchisq(.95, 2)), lty=2)
>
> m <- 300
> pb <- txtProgressBar(1, m, initial=1, style=3)
> points(xs[1, 1], xs[1, 2])
> for (i in 2:m){
+     Sys.sleep(0.1)
+     setTxtProgressBar(pb, i)
+     lines(xs[c(i, i - 1), ], col="gray")
+     points(xs[i, 1], xs[i, 2])
+ }
> close(pb)
```



When this animation is run interactively, and compared to the animation for the Metropolis algorithm, you can see how the Gibbs sampler is more efficient than the Metropolis algorithm in exploring the bivariate-normal density for the example.

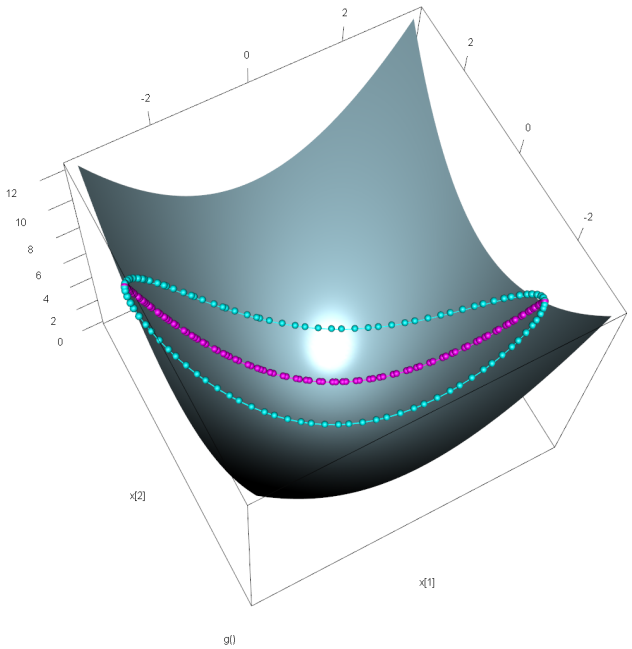
4 Hamiltonian Monte Carlo

Hamiltonian Monte Carlo (HMC), introduced to statistics by Neal (1996), is an improvement to the Metropolis-Hastings algorithm that, when properly tuned, provides more efficient proposals—proposals that are rejected less frequently but that also yield less autocorrelated samples from the target distribution. HMC is currently the state-of-the-art method of MCMC for sampling from continuous distributions.

4.1 Hamiltonian Dynamics

Hamiltonian Monte Carlo is named after the 19th Century physicist William Rowen Hamilton, who reformulated the mathematics of classical Newtonian mechanics. HMC exploits an analogy between exploring the surface of a probability density function and the motion of an object along a frictionless surface, propelled by its initial momentum and gravity.

Extending an example suggested by Neal (2011), think of a hockey puck (Neal's paper and this document were written in Canada after all) given a push in a particular direction on a frictionless and completely flat (horizontal) ice surface: The puck will continue to travel indefinitely in a straight line and with constant velocity in the direction in which it's pushed. Now imagine a surface that isn't flat, such as the one in the following figure:



This figure is the end result of a 100-step animation, in which two "pucks" (for convenience represented as small spheres) are released on the surface depicted above the point $\mathbf{x_0} = (-3.5, 2)'$, where x_1 and x_2 are the horizontal axes of the 3D space:

- The magenta puck is released with 0 momentum, and is therefore initially subject only to the force of gravity; the puck oscillates between the release point and an equally high point opposite to it.
- The cyan puck is released with a small momentum (0.5 and 1, respectively) in the positive directions of the x_1 and x_2 axes; this puck describes a more complex looping trajectory over the surface but also returns to its starting point.

Assuming, without any real loss of generality, that the puck has mass equal to 1,[1] at any instant, the potential energy of the puck due to gravity is equal to its height, while its kinetic energy is equal to the sum of the squared values of its momentum (mass $\times$ velocity $= 1 \times$ velocity) in the directions of x_1 and x_2. Because there is no loss of energy due to friction, conservation of energy dictates that the sum of potential energy and kinetic energy remains the same as the puck moves. When the puck moves downwards on the surface, its velocity increases, and hence its momentum and kinetic energy increase, while its height and hence its potential energy decrease; when the momentum of the puck carries it upwards on the surface, against gravity, the opposite is the case: momentum and kinetic energy decrease, height and potential energy increase.

By repeatedly introducing randomness into the momentum of the puck, HMC is able to visit the surface more generally, favouring lower regions of the surface. In a statistical application, the surface in question is the negative of the log of a multivariate probability density function (up to an additive constant on the log-density scale—i.e., a multiplicative constant on the density scale), and thus low regions correspond to regions of high probability. In the context of exploring a probability-density surface, the position variables X_1 and X_2 correspond to the random variables to be sampled, while the momentum variables are purely artificial, though necessary for the physical Hamiltonian analogy.

Metropolis proposals in HMC are more adapted to the probability surface to be sampled than in the traditional Metropolis or Metropolis-Hastings algorithms. By adjusting factors (discussed below) that affect the trajectory, it's possible to increase the proportion of accepted proposals and to decrease the autocorrelation of successively sampled values.

The surface in the illustration is for a bivariate-normal distribution with mean vector $\boldsymbol{\mu} = (0, 0)'$ and covariance matrix $\boldsymbol{\Sigma} = \mathbf{I}_2$ (the order-2 identity matrix)—that is, two uncorrelated standard-normal random variables. The graphed surface is the negative log-density of this bivariate-normal distribution, omitting the normalizing constant. Thus, the negative log-density surface differs from the graphed surface only by a constant difference in elevation, producing essentially the same dynamics for pucks sliding along both surfaces. The surface is a simple "bowl," whose vertical slices are parabolic and whose horizontal slices are circular, yielding the very simple dynamics illustrated in the figure.

The following R code drives the animation, the end state of which was depicted in the preceding figure; the animation is more effective, however, when viewed as it unfolds:

[1] The motion of the puck doesn't depends on its mass—recall Galileo and the Leaning Tower of Pisa—and setting mass to 1 simplifies the formulas given below

```
> hamTrajectory <- function(x0, mom0=c(0, 0), g, grad, stepsize=1,
+                           steps=100){
+     # x0: initial position
+     # mom0: initial momentum
+     # g: function to compute surface
+     # grad: function to compute gradient of surface
+     # stepsize: epsilon
+     # steps: number of steps
+     x <- x0 # position
+     xs <- matrix(0, steps + 1, 2)
+     xs[1, ] <- x
+     mom <- mom0 - 0.5*stepsize*grad(x0)  # half-step for momentum
+     for (step in 1:(steps - 1)){
+         x <- x + stepsize*mom
+         xs[step, ] <- x
+         mom <- mom - stepsize*grad(x)
+     }
+     xs[steps, ] <- x + stepsize*mom # last step for position
+     xs
+ }
>
> # log bivariate-normal density up to an additive contant
> gen_g <- function(mu, Sigma){
+     Sinv <- solve(Sigma)
+     # return a function of x
+     function(x) 0.5 * (x - mu) %*% Sinv %*% (x - mu)
+ }
>
> # derivatives
> gen_grad <- function(mu, Sigma){
+     Sinv <- solve(Sigma)
+     # return a function of x
+     function(x) as.vector(Sinv %*% (x - mu))
+ }
>
> # Example 1: uncorrelated, standard-normal variables, 0 initial momentum
> # releasing anywhere on the surface produces a simple oscillation
>
> mu <- c(0, 0)
> Sigma <- matrix(c(1, 0, 0, 1), 2, 2)
>
> g <- function(x, y) {
+     g <- gen_g(mu, Sigma)
+     n <- length(x)
```

```
+     z <- numeric(n)
+     for (i in 1:n){
+         z[i] <- g(c(x[i], y[i]))
+     }
+     z
+ }
>
> m <- 100  # steps
>
> persp3d(g, xlim=c(-3.5, 3.5), ylim=c(-3.5, 3.5), col="lightblue",
+         xlab="x[1]", ylab="x[2]", zlab="g()")
>
> xs1 <- hamTrajectory(c(-3.5, 2), mom0=c(0, 0), gen_g(mu, Sigma),
+                      gen_grad(mu, Sigma), stepsize=0.2, steps=m)
>
> pb <- txtProgressBar(1, m, initial=1, style=3)
> spheres3d(xs1[1, 1], xs1[1, 2], 0.1 + g(xs1[1, 1], xs1[1, 2]),
+           radius=0.1, col="magenta")
> for (i in 2:m){
+     Sys.sleep(0.05)
+     setTxtProgressBar(pb, i)
+     lines3d(xs1[c(i, i - 1), 1], xs1[c(i, i - 1), 2],
+             g(xs1[c(i, i - 1), 1], xs1[c(i, i - 1), 2]),
+             col="magenta")
+     spheres3d(xs1[i, 1], xs1[i, 2], 0.1 + g(xs1[i, 1], xs1[i, 2]),
+               radius=0.1, col = "magenta")
+ }
> close(pb)
>
> # Example 2: uncorrelated, standard-normal variables,
> #            nonzero initial momentum
> # releasing anywhere on the surface produces a closed curve
>
> xs2 <- hamTrajectory(c(-3.5, 2), mom0=c(0.5, 1), gen_g(mu, Sigma),
+                      gen_grad(mu, Sigma), stepsize=0.2, steps=m)
>
> pb <- txtProgressBar(1, m, initial=1, style=3)
> spheres3d(xs2[1, 1], xs2[1, 2], 0.1 + g(xs2[1, 1], xs2[1, 2]),
+           radius=0.1, col="cyan")
> for (i in 2:m){
+     Sys.sleep(0.05)
+     setTxtProgressBar(pb, i)
+     lines3d(xs2[c(i, i - 1), 1], xs2[c(i, i - 1), 2],
+             g(xs2[c(i, i - 1), 1], xs2[c(i, i - 1), 2]),
```

```
+              col="cyan")
+     spheres3d(xs2[i, 1], xs2[i, 2], 0.1 + g(xs2[i, 1], xs2[i, 2]),
+               radius=0.1, col = "cyan")
+ }
> close(pb)
>
> # save image
> rgl.snapshot("trajectory-1.png")
```

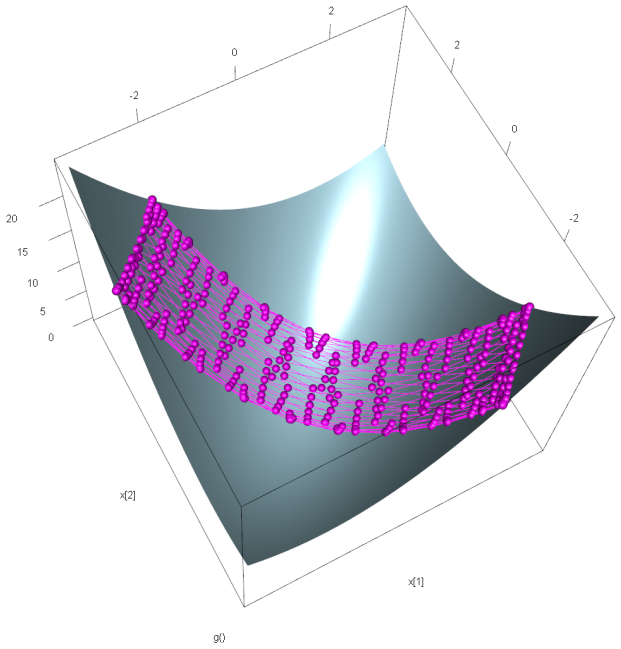
We'll explain the ideas underlying this code presently.

First, however, for a partly contrasting example, consider the surface shown in the following figure, generated by the bivariate normal distribution with mean vector $\boldsymbol{\mu} = (0, 0)'$ and covariance matrix

$$\boldsymbol{\Sigma} = \begin{pmatrix} 1 & 0.5 \\ 0.5 & 1 \end{pmatrix} \tag{10}$$

That is, X_1 and X_2 are standard-normal random variables with correlation $\rho = 0.5$. As in the first example, a puck is released with 0 momentum above the point $\mathbf{x} = (-3.5, 2)'$, and now 400 steps are shown, tracing out a much more elaborate trajectory than before. In this case, the "bowl" representing the negative log-density function is still parabolic in vertical cross-sections, but it is now elliptical in horizontal cross-sections, producing more complex dynamics.[2]

[2]Because the dynamics are unchanged by rotation of the surface around a vertical axis, the same effect can achieved be with uncorrelated Xs that have different standard deviations, stretching in the bowl in the direction of the larger standard deviation.



The code used to generate this figure is:

```
> # Example 3: correlated, standard-normal variables, 0 initial momentum
>
> mu <- c(0, 0)
> Sigma <- matrix(c(1, 0.5, 0.5, 1), 2, 2)
>
> g <- function(x, y) {
+     g <- gen_g(mu, Sigma)
+     n <- length(x)
+     z <- numeric(n)
+     for (i in 1:n){
+         z[i] <- g(c(x[i], y[i]))
+     }
+     z
+ }
>
> m <- 400
>
> persp3d(g, xlim=c(-3.5, 3.5), ylim=c(-3.5, 3.5), col="lightblue",
```

```
+         xlab="x[1]", ylab="x[2]", zlab="g()")
>
> xs3 <- hamTrajectory(c(-3.5, 2), mom0=c(0, 0), gen_g(mu, Sigma),
+                      gen_grad(mu, Sigma), stepsize=0.2, steps=m)
>
> pb <- txtProgressBar(1, m, initial=1, style=3)
> spheres3d(xs3[1, 1], xs3[1, 2], 0.1 + g(xs3[1, 1], xs3[1, 2]),
+           radius=0.2, col="magenta")
> for (i in 2:m){
+     Sys.sleep(0.05)
+     setTxtProgressBar(pb, i)
+     lines3d(xs3[c(i, i - 1), 1], xs3[c(i, i - 1), 2],
+             g(xs3[c(i, i - 1), 1], xs3[c(i, i - 1), 2]),
+             col="magenta")
+     spheres3d(xs3[i, 1], xs3[i, 2], 0.1 + g(xs3[i, 1], xs3[i, 2]),
+               radius=0.2, col = "magenta")
+ }
> close(pb)
>
> rgl.snapshot("trajectory-2.png")
```

In the general case with which we'll eventually be interested, the value of the surface giving the "elevation" of the puck is a function $g(\mathbf{x})$ of n "position" variables, comprising the vector $\mathbf{x}$. In a statistical application of HMC, the Xs are the random variables that we want to sample, and the height of the surface is the negative of the log of a function that may differ from the target density by a multiplicative constant. Thus, in the notation of Sections 2 and 3, $g(\mathbf{x}) = -\log p^*(\mathbf{x})$.

There are also n momentum variables, one for each of the dimensions of $\mathbf{x}$, in the vector $\mathbf{m}$. The Hamiltonian, $H(\mathbf{x}, \mathbf{m})$, is a function of $\mathbf{x}$ and $\mathbf{m}$, and, for the cases that we'll consider, is composed of two functions, which, in their physical interpretation, represent respectively potential and kinetic energy. Potential energy is equal to the elevation of the surface at the current position, $g(\mathbf{x})$, while kinetic energy, $k(\mathbf{m})$, is purely a function of momentation. The total energy of the puck is conserved as it moves, and so at any point in time t,

$$E = H[\mathbf{x}(t), \mathbf{m}(t)] = g[\mathbf{x}(t)] + k[\mathbf{m}(t)] \tag{11}$$

where

$$k(\mathbf{m}) = \tfrac{1}{2}\mathbf{m}'\mathbf{m} = \tfrac{1}{2}\sum_{i=1}^{n} m_i^2 \tag{12}$$

All this assumes, recall, that the mass of the puck is 1, which slightly simplifies the results (making, e.g., momentum equal to velocity).

The trajectory of the puck over "time," t, is given by Hamilton's equations

$$\begin{aligned} \frac{d\mathbf{m}}{dt} &= -\frac{\partial H(\mathbf{x},\mathbf{m})}{\partial \mathbf{x}} = -\frac{\partial g(\mathbf{x})}{\partial \mathbf{x}} \\ \frac{d\mathbf{x}}{dt} &= \frac{\partial H(\mathbf{x},\mathbf{m})}{\partial \mathbf{m}} = \mathbf{m} \end{aligned} \tag{13}$$

4.2 The Leapfrog Method

The time trajectory implied by Hamilton's differential equations (13) isn't in general solvable analytically, but the trajectory can be accurately approximated by discretizing time in small steps, ϵ, and applying the following algorithm (adapted from Neal, 2011), called the leapfrog method:[3]

- Start with values of $\mathbf{x}(0)$ and $\mathbf{m}(0)$ at time $t = 0$, and take a half-step for the momentum variables $\mathbf{m}$,

$$\mathbf{m}(\epsilon/2) = \mathbf{m}(0) - \frac{\epsilon}{2} \times \frac{\partial g(\mathbf{x})}{\partial \mathbf{x}}|_{\mathbf{x}=\mathbf{x}(0)} \tag{14}$$

- Then for $t = \epsilon, 2 \times \epsilon, \ldots, s \times \epsilon$ (where s is the number of steps), serially update the position and momentum variables, using the most recent value of each:

$$\begin{aligned} \mathbf{x}(t) &= \mathbf{x}(t-\epsilon) + \epsilon \times \mathbf{m}(t-\epsilon/2) \\ \mathbf{m}(t+\epsilon/2) &= \mathbf{m}(t-\epsilon/2) - \epsilon \times \frac{\partial g(\mathbf{x})}{\partial \mathbf{x}}|_{\mathbf{x}=\mathbf{x}(t)} \end{aligned} \tag{15}$$

  The time-length of the trajectory is thus $s \times \epsilon$.

The R function `hamTrajectory()`, given above, implements this method.

To apply the leapfrog methods to the illustrative bivariate normal distributions, we need the negative of the log density (ignoring the normalizing constant[4]) and its partial derivatives (the gradient), which are simply

$$\begin{aligned} g(\mathbf{x}) &= \tfrac{1}{2}(\mathbf{x}-\boldsymbol{\mu})'\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu}) \\ \frac{\partial g(\mathbf{x})}{\partial \mathbf{x}} &= \boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu}) \end{aligned} \tag{16}$$

These are coded respectively in the R functions `gen_g()` and `gen_grad()`, shown previously, which return functions of $\mathbf{x}$ given $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$.

[3] How good the approximation is depends on the step size and length of the trajectory.

[4] For the bivariate-normal distribution, the (ignored) multiplicative normalizing constant is $\left(2\pi\sqrt{\det\boldsymbol{\Sigma}}\right)^{-1}$.

4.3 HMC Sampling

To implement HMC sampling, the leapfrog method can be used to generate proposals to the Metropolis algorithm (Section 2).[5] To generate a proposal $\mathbf{x}^*$, we start at the current values $\mathbf{x}_i$ of the variables to be sampled, and randomly select the starting momentum in each direction—in the R code shown below, the momentum values are sampled independently from the standard-normal distribution, $N(0, 1)$. The Metropolis acceptance ratio (Equation 3 on page 3) becomes

$$a = \exp[H(\mathbf{x}_i, \mathbf{m}_i) - H(\mathbf{x}^*, \mathbf{m}^*)] \tag{17}$$

The acceptance ratio a depends on both the momentum variables $\mathbf{m}$ and the position variables $\mathbf{x}$, because both are necessary to characterize the current state of the system, even though only the position variables are of real interest.

If the leapfrog method were exact, then (because of conservation of energy) the energy for the proposal at the end of the path, $H(\mathbf{x}^*, \mathbf{m}^*)$, would be exactly equal to the energy at the beginning of the path, $H(\mathbf{x}_i, \mathbf{m}_i)$, in which case the acceptance ratio $a = \exp(0) = 1$, and the proposal would always be accepted. The acceptance ratio can only depart from 1 due to discretization error in the leapfrog method.[6] If we "tune" the step size and number of steps well, therefore, we should expect a high acceptance rate for proposals. To achieve both a high rate of acceptance and nearly independent draws from the target distribution, tuning is more critical for HMC than for simpler Metropolis sampling.

The following function (adapted from Neal, 2011) generates a proposal:

```
> hmcStep <- function(x0, g, grad, stepsize=1, steps=20){
+     # x0: initial position
+     # g: function to compute negative log of target
+     # grad: function to compute gradient of negative log of target
+     # stepsize: can be a vector
+     # steps: number of steps
+     x <- x0
+     mom0 <- rnorm(length(x)) # sample initial momentum values from N(0, 1)
+     mom <- mom0 - 0.5*stepsize*grad(x0)  # half-step for momentum
+     for (step in 1:(steps - 1)){ # all but last step
+         x <- x + stepsize*mom
+         mom <- mom - stepsize*grad(x)
+     }
+     x <- x + stepsize*mom # last step for position
```

[5] The leapfrog method is symmetric (i.e., reversable), and so we can use the simpler Metropolis algorithm instead of Metropolis-Hastings. In the code introduced below, we also permit different "step" sizes for the various elements of $\mathbf{x}$, effectively multiplying the time-increment ϵ by scaling factors for the position variables; this can be a useful approach when the variables have different scales (e.g., standard deviations). The resulting path along the surface isn't a true Hamiltonian trajectory, but it still provides legitimate update candidates to the Metropolis algorithm. A common alternative is to complicate the Hamiltonian equations by introducing a diagonal $n \times n$ "mass matrix" $\mathbf{M}$, the diagonal entries of which reflect the scales of the variables.

[6] Because, however, the leapfrog method is reversable, an error in approximating the Hamiltonian doesn't invalidate the method.

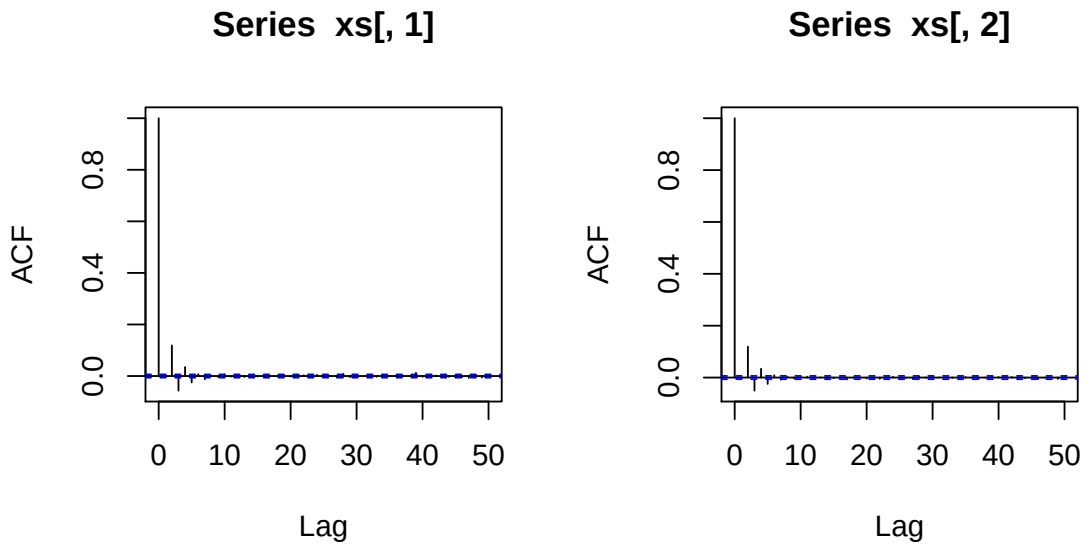
```
+     mom <- mom  - 0.5*stepsize*grad(x)  # last half-step for momentum
+     # evaluate proposal:
+     if (log(runif(1)) < g(x0) - g(x) + 0.5*sum(mom0^2)
+         - 0.5*sum(mom^2)) x else x0
+ }
```

We proceed to apply HMC to the bivariate-normal target distribution with

$$\begin{aligned} \boldsymbol{\mu} &= \begin{pmatrix} 1 \\ 2 \end{pmatrix} \\ \boldsymbol{\Sigma} &= \begin{pmatrix} 1 & 1 \\ 1 & 4 \end{pmatrix} \end{aligned} \tag{18}$$

used previously in Sections 2.1 and 3.1 to illustrate the Hastings and Gibbs algorithms:

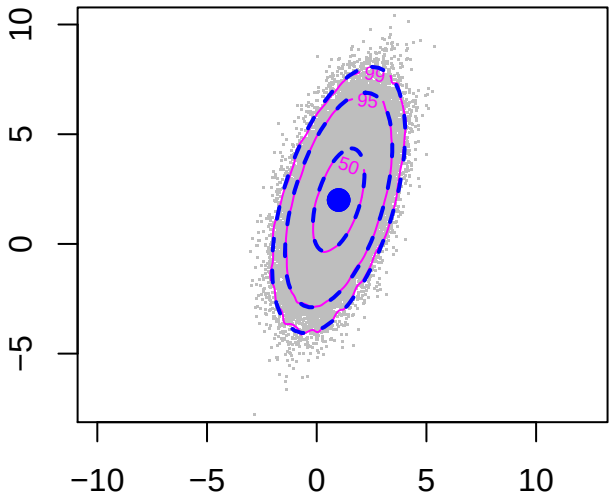
```
> mu <- c(1, 2)
> Sigma <- matrix(c(1, 1, 1, 4), 2, 2)
>
> accepted <- rejected <- 0
> x.current <- x.previous <- c(0, 0)
> m <- 1e5  # draws
> xs <- matrix(0, m, 2)  # to hold sampled values
>
> set.seed(305141)
>
> system.time(for (i in 1:m){
+     x.current <- hmcStep(x.previous, gen_g(mu, Sigma), gen_grad(mu, Sigma),
+                          stepsize=c(0.3, 0.6), steps=20)
+     if (all(x.current == x.previous)) rejected <- rejected + 1
+     else accepted <- accepted + 1
+     x.previous <- x.current
+     xs[i, ] <- x.current
+ })

   user  system elapsed
  16.91    0.00   16.90

> accepted/m

[1] 0.98693

> par(mfrow=c(1, 2))
> acf(xs[, 1])
> acf(xs[, 2])
```



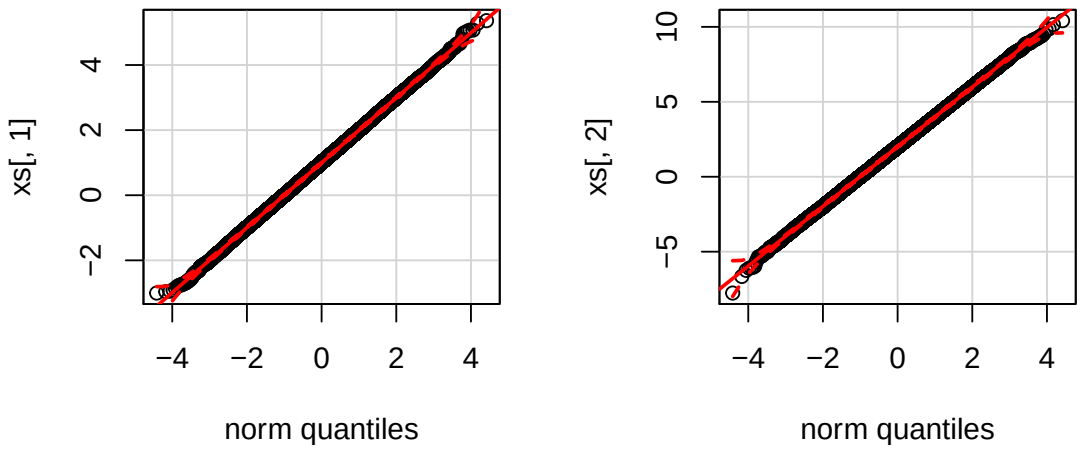
The proportion of accepted proposals is much higher than for the standard Metropolis version of this example in Section 2, and the HMC sampled values are much less autocorrelated. To achieve these desirable results, we had to select suitable values for the step sizes and number of steps by trial and error. In particular, step size was set twice as large in the direction of X_2 (which has standard deviation $\sigma_2 = 2$) than in the direction of X_1 (which has standard deviation $\sigma_1 = 1$.)

The HMC samples closely reproduce the target distribution:

```
> colMeans(xs)

[1] 1.000563 2.000666

> var(xs)

          [,1]     [,2]
[1,] 0.9965631 1.000904
[2,] 1.0009037 3.955792

> eqscplot(xs, pch=".", col="gray")
> res <- kde(xs)
> plot(res, add=TRUE, cont=c(50, 95, 99), col="magenta")
> ellipse(mu, Sigma, col='blue', radius=sqrt(qchisq(.5, 2)), lty=2)
> ellipse(mu, Sigma, col='blue', radius=sqrt(qchisq(.95, 2)), lty=2)
> ellipse(mu, Sigma, col='blue', radius=sqrt(qchisq(.99, 2)), lty=2)
```

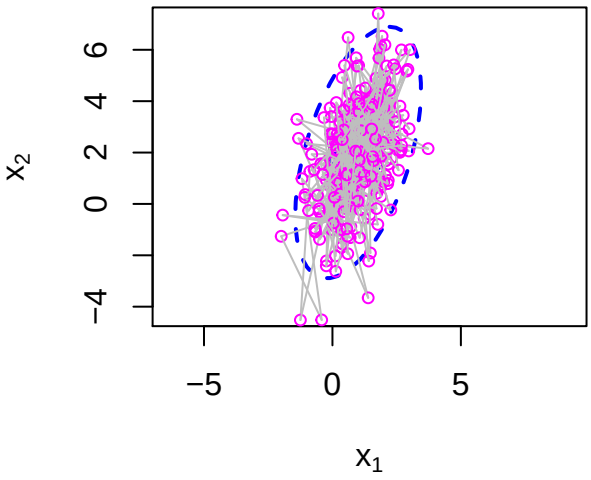


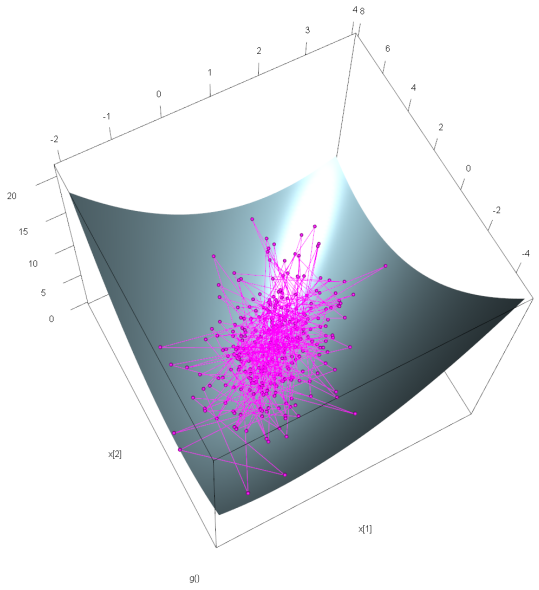

```
> par(mfrow=c(1, 2))
> qqPlot(xs[, 1])
> qqPlot(xs[, 2])
```



The following code generates animations in 2D and 3D of the first 300 sampled values of $\mathbf{x}$:

```
> m <- 300
>
> # 2D animation
>
> range <- c(min(apply(xs[1:300, ], 2, min)),
+            max(apply(xs[1:300, ], 2, max)))
> eqscplot(0, 0, type="n", xlim=range, ylim=range,
+          xlab=expression(x[1]), ylab=expression(x[2]))
> ellipse(mu, Sigma, col='blue', radius=sqrt(qchisq(.95, 2)), lty=2)
>
> pb <- txtProgressBar(1, m, initial=1, style=3)
> points(xs[1, 1], xs[1, 2])
> for (i in 2:m){
+     Sys.sleep(0.1)
+     setTxtProgressBar(pb, i)
+     lines(xs[c(i, i - 1), ], col="gray")
+     points(xs[i, 1], xs[i, 2],
+            col = if(all(xs[i, ] == xs[i - 1,])) "blue" else "magenta")
+ }
> close(pb)
>
> # 3D animation
>
> persp3d(g, xlim=c(-2, 4), ylim=c(-5, 8), col="lightblue",
+         xlab="x[1]", ylab="x[2]", zlab="g()")
>
> pb <- txtProgressBar(1, m, initial=1, style=3)
> spheres3d(xs[1, 1], xs[1, 2], 0.1 + g(xs[1, 1], xs[1, 2]),
+           radius=0.1, col="magenta")
> for (i in 2:m){
+     Sys.sleep(0.05)
+     setTxtProgressBar(pb, i)
+     lines3d(xs[c(i, i - 1), 1], xs[c(i, i - 1), 2],
+             g(xs[c(i, i - 1), 1], xs[c(i, i - 1), 2]), col="magenta")
+     spheres3d(xs[i, 1], xs[i, 2], 0.1 + g(xs[i, 1], xs[i, 2]), radius=0.1,
+               col = if (all(xs[i, ] == xs[i - 1, ])) "blue" else "magenta")
+ }
> close(pb)
>
> rgl.snapshot("HMC.png")
```





In both graphs, magenta points represent accepted proposals and blue points rejected proposals.[7]

[7] In the final displayed graphs, the rejected proposals are so few that the blue points aren't discernable.

5 A Simple Application to Bayesian Inference

We'll illustrate the application of MCMC to Bayesian inference by considering a simple single-parameter problem: estimating a probability (or population proportion).

Consider the following example (extending an example in Fox, 2009, Sec. 3.7.3): We're interested in the probability π of obtaining a head in a flip of a particular coin. We're not very patient, and so we collect data by flipping the coin only $n = 10$ times, being careful, however, to shake up the coin between flips and to flip it irregularly so that it's credible that the flips are statistically independent. In the event, our 10 flips produce the following sequence of 7 heads (H) and 3 tails (T): HHTHHHTTHH.

Assuming independent flips and constant probability π of a head leads to the Bernoulli likelihood[8]

$$L(\pi|\text{HHTHHHTTHH}) = \pi\pi(1-\pi)\pi\pi\pi(1-\pi)(1-\pi)\pi\pi = \pi^7(1-\pi)^3 \tag{19}$$

The generalization of this example is for n flips of the coin, observing h heads and $n-h$ tails, in which case the Bernoulli likelihood is $L(\pi|\text{data}) = \pi^h(1-\pi)^{n-h}$.

The conjugate prior for the Bernoulli likelihood is the beta distribution $\text{Beta}(a, b)$, with density function

$$p(\pi; a, b) = \frac{\pi^{a-1}(1-\pi)^{b-1}}{B(a,b)} \text{ for } 0 \leq \pi \leq 1 \text{ and } a, b \geq 0 \tag{20}$$

where $B(a, b)$ is the normalizing constant for the beta distribution,

$$\begin{aligned} B(a,b) &= \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)} \\ \Gamma(v) &= \int_0^\infty e^{-v} z^{v-1} dz \end{aligned} \tag{21}$$

The beta distribution is the conjugate prior because the resulting posterior is also beta:

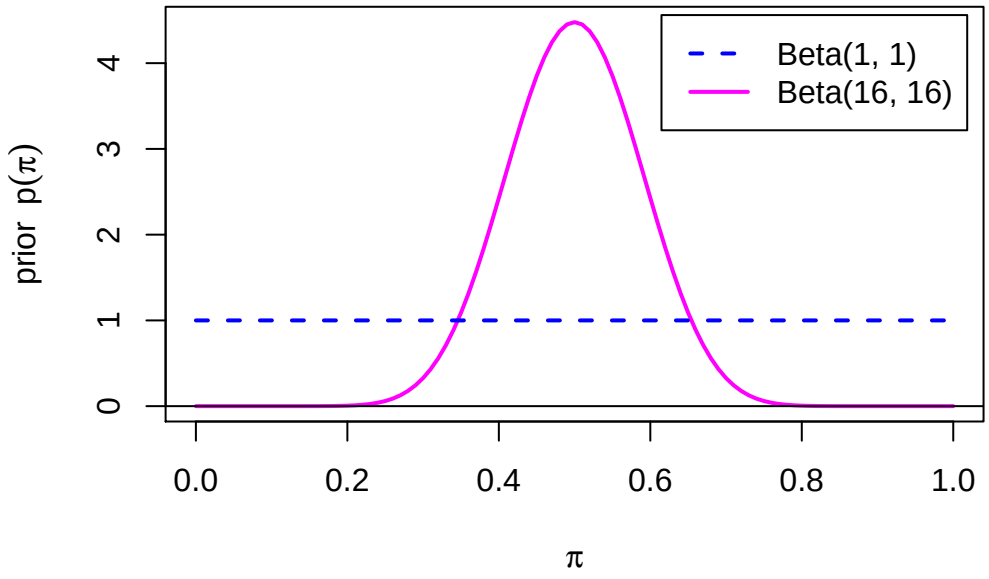
$$\begin{aligned} p(\pi|\text{data}) &\propto L(\pi|\text{data}) \times p(\pi; a, b) \\ &= \pi^h(1-\pi)^{n-h} \times \pi^{a-1}(1-\pi)^{b-1} \\ &= \pi^{h+a-1}(1-\pi)^{n-h+b-1} \end{aligned} \tag{22}$$

That is, $p(\pi|\text{data}) = \text{Beta}(h + a, n - h + b)$.

The trick then in applying this result is to pick values a and b so that the prior $\text{Beta}(a, b)$ reflects initial uncertainty in the value of π. For example, picking $a = b = 1$ produces a flat prior for π, while picking $a = b = 16$ produces an informative prior that confines the prior density to be quite close to $\pi = 0.5$, as might be appropriate if examination of the coin led us to believe that it is fair:

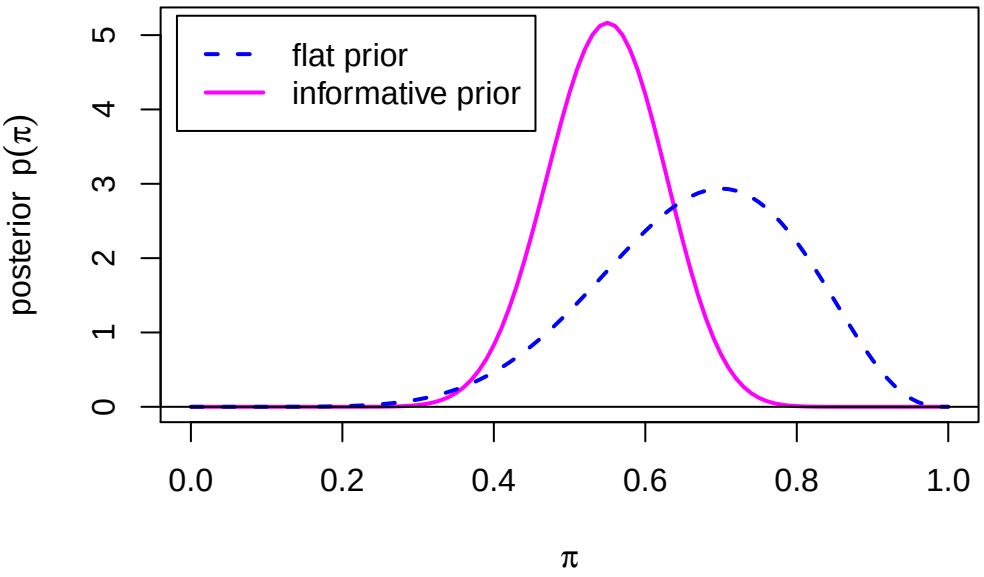
[8] Because, as is evident, the likelihood doesn't depend on the order of heads and tails, only on their numbers, the number of heads (for the fixed number of flips $n = 10$) is a sufficient statistic for the parameter π, and we could equivalently work with the binomial distribution $\text{Bin}(H; n = 10)$ for the number of heads H. The likelihood for the example would then be $L(\pi|\text{data}) = \frac{10!}{7!3!}\pi^7(1-\pi)^3$, which produces the same inference about π.

```
> curve(dbeta(x, shape1=16, shape2=16),
+       0, 1, col="magenta", lwd=2,
+       xlab=expression(pi), ylab=expression(prior~~p(pi)))
> curve(dbeta(x, shape1=1, shape2=1),
+       0, 1, col="blue", lwd=2, lty=2, add=TRUE)
> legend("topright", legend=c("Beta(1, 1)", "Beta(16, 16)"),
+        col=c("blue", "magenta"), lty=c(2, 1), lwd=2, inset=0.02)
> abline(h=0)
```



The posteriors that correspond to these two priors are therefore, respectively, Beta$(8, 4)$ and Beta$(23, 19)$:

```
> curve(dbeta(x, shape1=23, shape2=19),
+       0, 1, col="magenta", lwd=2,
+       xlab=expression(pi), ylab=expression(posterior~~p(pi)))
> curve(dbeta(x, shape1=8, shape2=4),
+       0, 1, col="blue", lwd=2, lty=2, add=TRUE)
> legend("topleft", legend=c("flat prior", "informative prior"),
+        col=c("blue", "magenta"), lty=c(2, 1), lwd=2, inset=0.02)
> abline(h=0)
```



In the case of the flat prior, the posterior is simply proportional to the likelihood, and thus is at its maximum at the MLE of π, $\widehat{\pi} = 7/10 = 0.7$ (the sample proportion of heads).

Imagine, however, that we don't know that the beta prior and Bernoulli likelihood combine to produce a simple beta posterior, and instead use MCMC to approximate the posterior. Because we want to try a few variations, we'll write some functions for the Metropolis algorithm rather than relying on a script, as we did in Section 2. Moreover, because it doesn't require much additional effort, we'll aim for some generality, allowing the user to specify the likelihood function, prior distribution, and proposal distribution, and permitting the parameter to be a vector. We'll also have our `metropolis()` function return an object (of class `"metropolis"`) as its result, making the result a bit easier to manipulate:

```
> metropolis <- function(likelihood, prior, proposal, pars0, m=1e5){
+     # likelihood: a function of the parameter vector
+     # prior:      a function of the parameter vector
+     # proposal:   a function of the current value of the parameter
+     #              vector, returning a proposed parameter vector
+     # pars0:      a vector of initial values of the parameters
+     # m:          number of samples, defaults to 10^5
+     post <- function(pars) likelihood(pars)*prior(pars)
+     pars <- matrix(0, m, length(pars0))
+     accepted <- rejected <- 0
+     pars.current <- pars0
```

```
+     for (i in 1:m){
+         pars.proposed <- proposal(pars.current)
+         a <- post(pars.proposed)/post(pars.current)  # aceptance ratio
+         if (a >= 1 || runif(1) <= a) {
+             pars[i, ] <- pars.proposed
+             pars.current <- pars.proposed
+             accepted <- accepted + 1
+         }
+         else {
+             pars[i, ] <- pars.current
+             rejected <- rejected + 1
+         }
+     }
+     if (length(pars0) == 1) pars <- as.vector(pars)
+     result <- list(samples=pars, accepted=accepted, rejected=rejected,
+                    thinned=FALSE)
+     class(result) <- "metropolis"
+     result
+ }
>
> print.metropolis <- function(x, ...){
+     cat("number of samples: ",
+         with(x, if (is.matrix(samples)) nrow(samples)
+              else length(samples)))
+     cat("\nnumber of parameters: ",
+         with(x, if (is.matrix(samples)) ncol(samples) else 1))
+     if (x$thinned){
+         cat("\nPrior to thinning:")
+     }
+     cat("\nnumber of proposals accepted: ", x$accepted)
+     cat("\nnumber of proposals rejected: ", x$rejected)
+     cat("\npercentage of proposals accepted: ",
+         with(x, round(100*accepted/(accepted + rejected), 2)), "\n")
+     if (is.matrix(x$samples)){
+         cat("\nestimated posterior medians")
+         print(apply(x$samples, 2, median))
+     }
+     else cat("\nestimated posterior median: ", median(x$samples))
+     invisible(x)
+ }
>
> plot.metropolis <- function(x, ...){
+     n.par <- if (is.matrix(x$samples)) ncol(x$samples) else 1
+     if (n.par == 1) acf(x$samples, main="", ...)
```

```
+     else{
+         save.mfrow <- par(mfrow =  n2mfrow(n.par))
+         on.exit(options(save.mfrow))
+         for (j in 1:npar){
+             acf(x$samples[, j], main = paste("parameter", j), ...)
+         }
+     }
+ }
>
> thin <- function(object, ...){
+     UseMethod("thin")
+ }
>
> thin.metropolis <- function(object, by, ...){
+     # by: every by-th sample in object is retained
+     samples <- object$samples
+     object$samples <- if (is.matrix(samples)){
+         samples[seq(1, nrow(samples), by=by), ]
+     }
+     else samples[seq(1, length(samples), by=by)]
+     object$thinned <- TRUE
+     object
+ }
```

The `print()` method supplied for `"metropolis"` objects briefly summarizes the results, the `plot()` method graphs the autocorrelation functions of the sampled parameter values, and the `thin()` method (for a new `thin()` generic) thins the samples in a `"metropolis"` object. A caveat: These functions haven't been tested extensively.

Additionally, we'll write some R functions to generate the Bernoulli likelihood function and beta prior:

```
> # Bernoulli likelihood
> L <- function(n, h) {
+     function(pi) pi^h * (1 - pi)^(n - h)
+ }
>
> # conjugate Beta prior
> prior <- function(a, b) {
+     function(pi) dbeta(pi, a, b)
+ }
```

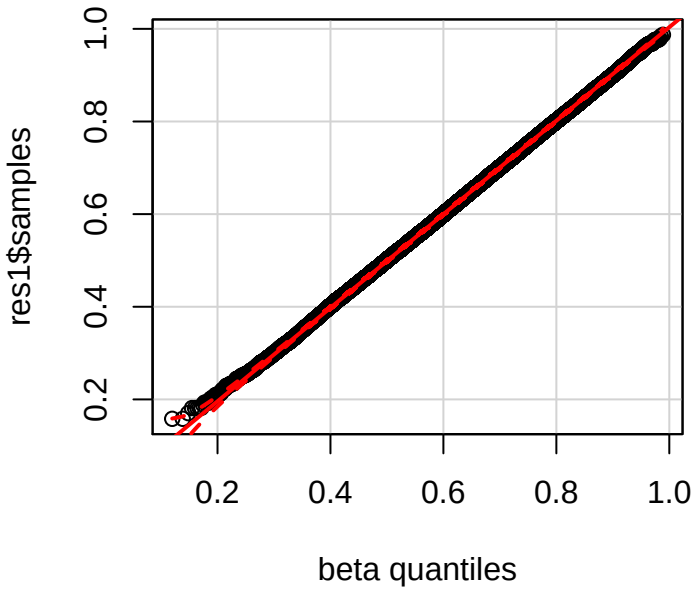
Thus, `L()` takes the sample size and obtained number of heads (in our example, $n = 10$ and $h = 7$) as arguments and returns the likelihood as a function of the probability of a head π, while `prior()` takes the shape parameters a and b of the beta function as arguments and returns the corresponding beta density function of π.

For our first simulation, we specify a flat prior ($a = b = 1$), and set the standard deviation of the normal proposal distribution to 0.1, generating 10^5 samples from the posterior distribution of π:

```
> set.seed(687431)  # for reproducibility
>
> res1 <- metropolis(
+     likelihood = L(10, 7),
+     prior = prior(1, 1),
+     proposal = function (par) rnorm(1, mean=par, sd=0.1),
+     pars0 = 0.5
+ )
> res1

number of samples:  100000
number of parameters:  1
number of proposals accepted:  77484
number of proposals rejected:  22516
percentage of proposals accepted:  77.48

estimated posterior median:  0.6768293
```

Let's check the distribution of the obtained samples against the right answer, Beta$(8, 4)$, using a theoretical quantile-comparison (QQ) plot (provided by the `qqPlot()` function in the **car** package):

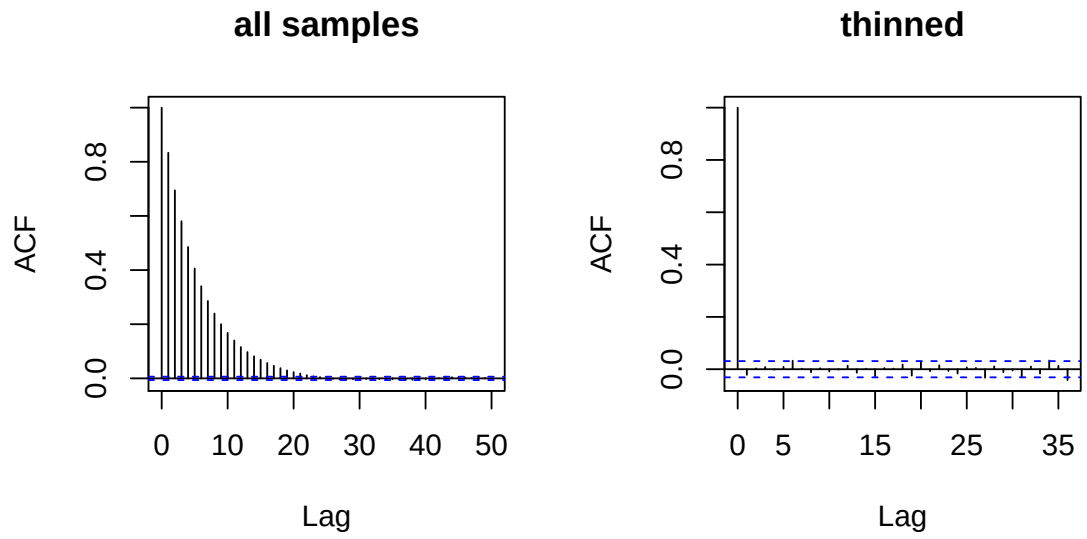
```
> qqPlot(res1$samples, dist="beta", shape1=8, shape2=4)
```



The broken red lines on the QQ plot mark off a point-wide 95% confidence envelope, assuming an independent sample from the beta distribution (which, as we'll shortly see, is a poor assumption here, making the envelope unrealistically narrow). The result looks quite good, although the lower tail of the sampled values appears to be a bit short. The sampled values are, however, substantially autocorrelated, with the autocorrelations dying out by lag 25:

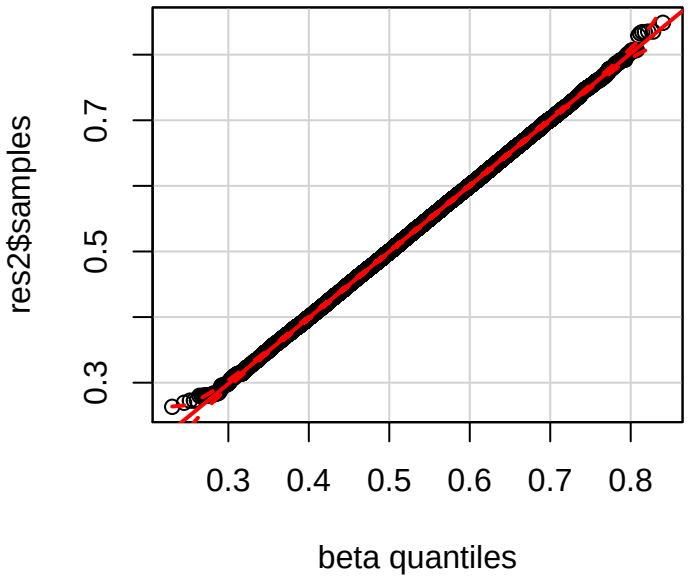
```
> par(mfrow=c(1, 2))
>
> plot(res1)
> title(main="all samples")
>
> res1thin <- thin(res1, 25)
> res1thin

number of samples:  4000
number of parameters:  1
Prior to thinning:
number of proposals accepted:  77484
number of proposals rejected:  22516
percentage of proposals accepted:  77.48

estimated posterior median:  0.6777007

> plot(res1thin)
```

```
> title(main="thinned")
```



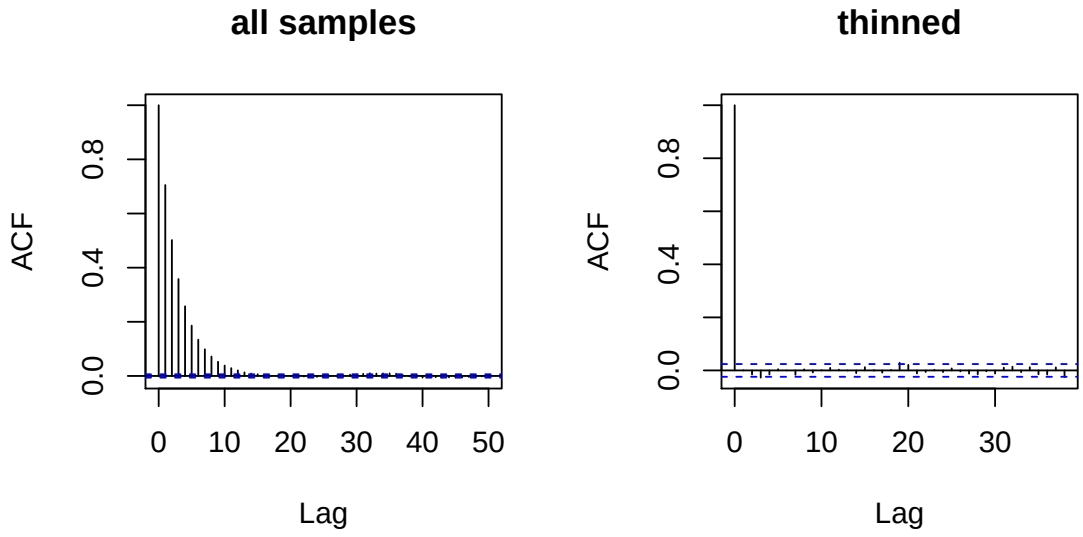
Next, we'll repeat this example for the informative Beta(16, 16) prior, for which the posterior should be Beta(23, 19):

```
> set.seed(377744)  # for reproducibility
>
> res2 <- metropolis(
+     likelihood = L(10, 7),
+     prior = prior(16, 16),
+     proposal = function (par) rnorm(1, mean=par, sd=0.1),
+     pars0 = 0.5
+ )
> res2

number of samples:  100000
number of parameters:  1
number of proposals accepted:  63236
number of proposals rejected:  36764
percentage of proposals accepted:  63.24

estimated posterior median:  0.5488176

> qqPlot(res2$samples, dist="beta", shape1=23, shape2=19)
```



In this case, the sampled values are somewhat less autocorrelated, and we thin them by taking every 15th value:

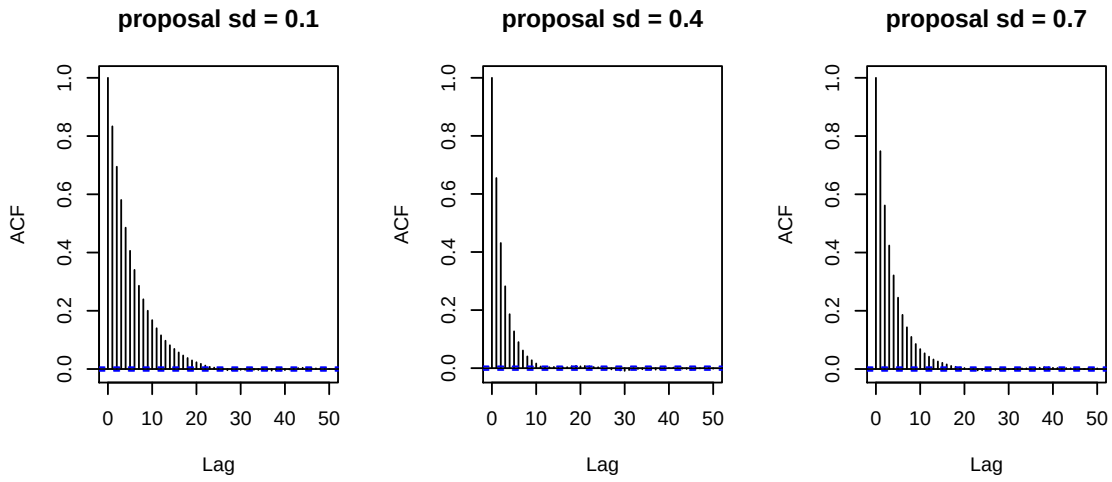
```
> par(mfrow=c(1, 2))
>
> plot(res2)
> title(main="all samples")
>
> res2thin <- thin(res2, 15)
> res2thin

number of samples:  6667
number of parameters:  1
Prior to thinning:
number of proposals accepted:  63236
number of proposals rejected:  36764
percentage of proposals accepted:  63.24

estimated posterior median:  0.5493089

> plot(res2thin)
> title(main="thinned")
```



Finally, returning to the flat prior, we'll illustrate how making the standard deviation of the proposal distribution larger (it's set initially, recall, to 0.1) can, up to a point, decrease the autocorrelation of the sampled values, reducing the need for thinning:

```
> set.seed(114724)
>
> res1a <- metropolis(
+     likelihood = L(10, 7),
+     prior = prior(1, 1),
+     proposal = function (par) rnorm(1, mean=par, sd=0.4),
+     pars0 = 0.5
+ )
>
> res1b <- metropolis(
+     likelihood = L(10, 7),
+     prior = prior(1, 1),
+     proposal = function (par) rnorm(1, mean=par, sd=0.7),
+     pars0 = 0.5
+ )
>
> res1

number of samples:  100000
number of parameters:  1
number of proposals accepted:  77484
number of proposals rejected:  22516
percentage of proposals accepted:  77.48
```

```
estimated posterior median:  0.6768293

> res1a

number of samples:  100000
number of parameters:  1
number of proposals accepted:  37286
number of proposals rejected:  62714
percentage of proposals accepted:  37.29

estimated posterior median:  0.6763135

> res1b

number of samples:  100000
number of parameters:  1
number of proposals accepted:  22997
number of proposals rejected:  77003
percentage of proposals accepted:  23

estimated posterior median:  0.6762038

> par(mfrow=c(1, 3))
> plot(res1)
> title(main="proposal sd = 0.1")
> plot(res1a)
> title(main="proposal sd = 0.4")
> plot(res1b)
> title(main="proposal sd = 0.7")
```



Notice as well how the acceptance rate of proposals declines as the standard deviation of

the proposal distribution increases: Because the Metropolis algorithm spends more time in high-density regions of the target distribution than in low-density regions, taking larger steps away from current parameter values tends to decrease the density at proposed values, consequently decreasing the probability of acceptance. In this example, setting the standard deviation of the proposal distribution to 0.4 produces lower autocorrelation in the sampled values than do standard deviations of 0.1 or 0.7.

References

G. Casella and E. J. George. Explaining the Gibbs sampler. *The American Statistician*, 46 (3):167–174, 1992.

S. Chib and E. Greenberg. Understanding the Metropolis-Hastings algorithm. *The American Statistician*, 49(4):327–335, 1995.

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

A. E. Gelfand and A. F. M. Smith. Sampling-based approaches to calculating marginal densities. *Journal of the American Statistical Association*, 85:398–409, 1990.

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

S. Geman and D. German. Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6:721–742, 1984.

W. K. Hastings. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika*, 57:97–109, 1970.

N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equation of state calculations by fast computing machines. *Journal of Chemical Physics*, 21:1087–1092, 1953.

R. M. Neal. *Bayesian Learning for Neural Networks*. Springer, New York, 1996.

R. M. Neal. MCMC using Hamiltonian dynamics. In S. Brooks, A. Gelman, G. L. Jones, and X.-L. Meng, editors, *Handbook of Markov Chain Monte Carlo*, chapter 5, pages 113–162. Chapman & Hall/CRC Press, Boca Raton FL, 2011.