Markov Chain Monte Carlo techniques

Finnian Bender

May 10, 2019

1 Introduction

Monte Carlo methods can be used for a variety of tasks for which exact simulation would be computationally infeasible. Often times, the underlying random variable $\pi(x)$ will be difficult, if not impossible, to sample directly because of high dimensionality or extreme complexity. In these scenarios, one powerful family of techniques that can be brought to bear are Markov Chain Monte Carlo methods. These methods, in particular the Metropolis-Hastings algorithm which will be expanded on later, allow us to sample from any distribution, regardless of their complexity or number of dimensions. Of course, these methods are not without their drawbacks, but with some extra work these drawbacks can be mitigated.

2 Markov Chains

We will begin by defining several key concepts that will be referenced throughout.

Definition 2.1. A *stochastic (random) process* $X := \{X_t | t \in T\}$ is a family of random variables X_t where T is an index set, generally called *time* regardless of if it is a time-based process, and $X_t : \Omega \to S$, where S is the *state space* of the process.

Definition 2.2. The *transition probabilities* $\mathbb{P}(X_1 = j | X_0 = i_0)$ give the probabilities that a process that is in state i_0 at time 0 will be in state j at time 1. The transition probabilities may depend on multiple previous times, e.g. $\mathbb{P}(X_2 = k | X_1 = j, X_0 = i_0)$

One way of thinking about this property is that the process's future depends only on its present, and not on its past.

Definition 2.3. A process is said to have the *Markov property* if $\mathbb{P}(X_{n+1} = j|X_n = i_n, ...X_0 = i_0) = \mathbb{P}(X_{n+1} = j|X_n = i_n)$

Definition 2.4. A *Markov chain* is a discrete-time stochastic process that has the Markov property.

Definition 2.5. A *(1-step)transition probability matrix* $\underline{\underline{P}}$ is a matrix whose (i,j) entry gives the probability that a Markov chain in state i will move to state j on its next move, called the *transition probability* $p_{i,j}$. Such a matrix is called *stochastic* if the sum of the rows is 1. For ease of notation, we will often write the (i,j) element as P(i,j).

Definition 2.6. An *n-step transition probability* $p^n_{i,j}$ is the probability that, given the chain is in state i at time 0, the chain will be in state j at time n. The *n-step transition probability matrix* $\underline{\underline{P^n}}$ is the matrix whose (i,j) entry is $p^n_{i,j}$

Theorem 1 (Chapman-Kolmogorov)**.** $(\underline{\underline{P}})^n = \underline{\underline{P^n}}$

Definition 2.7. In the case where $n \to \infty$, the rows of $\underline{\underline{P^n}}$ go towards a single distribution $\underline{\pi} = (...\pi_i...)$, where $\underline{\pi} = \underline{\pi}\ \underline{\underline{P}}$. Additionally, $\sum_i \pi(i)p_{i,j} = \pi_j$ and $\sum_i \pi(i) = 1$. Such a $\underline{\pi}$ is called the *stationary distribution* of the Markov chain represented by $\underline{\underline{P}}$. In general, we will write $\underline{\pi}$ as a function $\pi(i)$.

Probabilistically, this can be understood to mean that for large enough n, the n^{th} step of the Markov chain will be in state i with probability $\pi(i)$.

At this point, we have enough to begin outlining the prototypical Markov Chain Monte Carlo method, the Metropolis-Hastings algorithm.

3 The Metropolis-Hastings Algorithm

We will begin with a general description of what the algorithm looks like before providing a more rigorous definition of how it proceeds. Assume that there is some probability density $\pi(x)$ that we wish to sample but are unable to sample directly. Then we will create a Markov chain whose stationary distribution is the desired $\pi(x)$. To do this, we will start with some arbitrary distribution f(x). Then, we pick an arbitrary point in the state space, call it i, to begin our chain. Next, we pick a state in the state space that is slightly different from our starting point, call it j, and propose that our chain move to this state. We will the flip a coin whose probability of success is the ratio $\pi(j)f(j|i)/\pi(i)f(i|j)$. If the flip succeeds, we move to j. If not, we stay at i.

After repeating this for a large number of iterations, the probability that we will be in any given state is $\pi(i)$, regardless of what the initial distribution f(x) was.

Now we will provide a more rigorous definition of this method, as well as justification for why it works. First, take $\pi(x)$ to be our desired pdf. Then let J(i,j) be the transition probability matrix of a Markov chain on the same state space as $\pi(x)$. Now we wish to transform J(i,j) into a Markov chain K(i,j) with stationary distribution $\pi(i)$. First, we will define the acceptance function, acc(i,j), as follows:

$$\text{acc}(i,j) = \frac{\pi(j)J(j,i)}{\pi(i)J(i,j)}$$

where $\text{acc}(i,j)$ gives the probability that a proposed move from i to j will be accepted.

Remark. Note that often it is convenient to pick a Markov chain with a symmetric transition probability matrix, i.e. one in which J(i,j)=J(j,i) $\forall$ i,j. In this case, we can simplify the acceptance function to instead be

$$\text{acc}(i,j) = \frac{\pi(j)}{\pi(i)}$$

Now define K(i,j) as follows:

$$K(i,j) = \begin{cases} J(i,j) & i \neq j, acc(i,j) \geq 1 \\ J(i,j)\text{acc}(i,j) & i \neq j, acc(i,j) < 1 \\ J(i,j) + \sum_{k:acc(x,k)<1} (J(i,k)(1-acc(i,k)) & i = j \end{cases} \tag{1}$$

It is not immediately obvious why the Markov chain produced by this method will have a stationary distribution of π as we require. Rather than attempt to directly show that $\sum_i \pi(i)K(i,j) = \pi(j)$, we will instead prove that K satisfies the stronger condition given by the *detailed balance equation*, which states that

$$\pi(i)K(i,j) = \pi(j)K(j,i)$$

as it is trivial to show that the detailed balance equation implies that $\sum_i \pi(i)K(i,j) = \pi(j)$.

Theorem 2. The Markov chain whose transition probability matrix is defined in (1) satisfies the detailed balance equation.

Proof. First, note that if i=j, then the detailed balance condition is met trivially as

$$\pi(i)K(i,i) = \pi(i)K(i,i)$$

regardless of the value of i. Now, we look at the case when $i \neq j$. Note that we can reformulate the first two cases in our definition of K(i,j) to instead be that if i=j, then

$$K(i,j) = J(i,j)\min\{1, acc(i,j)\}$$

From this, we have that

$$\begin{aligned} \pi(i)K(i,j) &= \pi(i)J(i,j)\min\{1, \mathrm{acc}(i,j)\} \\ &= \pi(i)J(i,j)\min\left\{1, \frac{\pi(j)J(j,i)}{\pi(i)J(i,j)}\right\} \\ &= \min\{\pi(i)J(i,j), \pi(j)J(j,i)\} \end{aligned} \tag{2}$$

A similar argument gives us that

$$\pi(j)K(j,i) = \min\{\pi(j)J(j,i), \pi(i)J(i,j)\} \tag{3}$$

Since the minimum function is symmetric, i.e., min{x,y}=min{y,x}, equations (2) and (3) give the same result, so

$$\pi(i)K(i,j) = \pi(j)K(j,i)$$

and thus the detailed balance equation holds ∀ i,j in the state space of the Markov chain K. □

4 An Implementation of MCMC: Integration

One useful implementation of Markov Chain Monte Carlo is in the estimation of integrals. Let $f : \mathbb{R} \to \mathbb{R}$, $a, b \in \mathbb{R}$. To estimate the integral $F = \int_a^b f(x)dx$, we will take N independent and identically distributed (iid) uniform random variables $X_1, ..., X_N$ with $X_i \in [a, b]$ so the random variables have pdf$(X_i) = \frac{1}{b-a}$. Then we define the *Monte Carlo estimator* of f(x) as

$$\langle F^N \rangle := (b-a)\frac{1}{N}\sum_{i=0}^{N-1} f(X_i)$$

Theorem 3. $\mathbb{E}[\langle F^N \rangle] = F$

Proof.

$$\mathbb{E}[\langle F^N \rangle] = \mathbb{E}\left[(b-a)\frac{1}{N}\sum_{i=0}^{N-1} f(X_i)\right] = \frac{b-a}{N}\mathbb{E}\left[\sum_{i=0}^{N-1} f(X_i)\right]$$

Then, since the X_i are iid, the expectation splits across sums and each summand will have equal value, so we get that

$$\mathbb{E}[\langle F^N \rangle] = \frac{b-a}{N}(N\mathbb{E}[f(X_i)]) = (b-a)\mathbb{E}[f(X_i)]$$

$$= (b-a)\int_a^b f(x)pdf(X_i)dx = (b-a)\int_a^b f(x)\frac{1}{b-a}dx$$

$$= \int_a^b f(x)dx = F$$

Thus, $\mathbb{E}[\langle F^N \rangle] = F$ as required. □

In fact, this theorem is a special case of a more general theorem that works for multidimensional functions and random variables other than uniform random variables. We will redefine the *Monte Carlo estimator* to instead be given by the following:

$$\langle F^N \rangle = \frac{1}{N}\sum_{0}^{N-1} \frac{f(X_i)}{pdf(X_i)}$$

Then, we can state the more generalized version of Theorem 3:

Theorem 4. $\mathbb{E}[\langle F^N \rangle] = F$

While the proof is similar to the proof for Theorem 3, if flows much more smoothly.

Proof. Let S be our state space. Then

$$\mathbb{E}[\langle F^N \rangle] = \mathbb{E}\left[\frac{1}{N}\sum_{0}^{N-1} \frac{f(X_i)}{pdf(X_i)}\right] = \frac{1}{N}\sum_{0}^{N-1}\int_S \frac{f(X_i)}{pdf(X_i)}pdf(X_i)dx$$

$$= \int_S f(x)dx = F$$

Thus $\mathbb{E}[\langle F^N \rangle] = F$ as required. □

Thanks to the Strong Law of Large Numbers, $\mathbb{P}(\lim_{x\to\infty} X_i = \mathbb{E}[X]) = 1$, so if we take N to be large enough we will eventually converge to the right answer. At this point, a reasonable question would be to ask what the variance or standard deviation of the estimator is. This is fairly straightforward to calculate, once again due to the iid nature of our random variables.

$$\begin{aligned}\sigma^2[\langle F^N\rangle] &= \sigma^2\left[\frac{1}{N}\sum_0^{N-1}\frac{f(X_i)}{pdf(X_i)}\right] = \frac{1}{N^2}\sigma^2\left[\sum_0^{N-1}\frac{f(X_i)}{pdf(X_i)}\right] \\ &= \frac{1}{N^2}\sum_0^{N-1}\sigma^2\left[\frac{f(X_i)}{pdf(X_i)}\right] = \frac{1}{N}\sigma^2\left[\frac{f(X_i)}{pdf(X_i)}\right]\end{aligned} \tag{4}$$

Let $Y_i = \frac{f(X_i)}{pdf(X_I)}$. Then we can write our variance as $\sigma^2[\langle F^N\rangle] = \frac{1}{N}\sigma^2[Y_i]$, or in terms of standard deviation we write

$$\sigma = \frac{1}{\sqrt{N}}\sigma[Y_i]$$

Note that this implies that the variance from using this method of estimating our random variable does not depend on the number of dimensions of our integral as all of the terms are independent of the number of dimensions. From this we can see that there are two main methods of decreasing the amount of error in our estimator. Either we can increase the number of random variables, or we can try to reduce the variance of Y_i by choosing a random variable whose pdf resembles a scaled version of the function we are integrating. In the best case, we would pick our random variable such that $pdf(X_i) = \alpha f(x)$, where α is some constant, In this case, we would find that

$$\sigma^2[Y_i] = \sigma^2[\frac{f(X_i)}{\alpha f(X_i)}] = \sigma^2[\alpha] = 0$$

Thus, if you pick your random variable such that it is directly proportional to the function of integration, there will be zero variance! However, in practice, we cannot pick such a random variable, as

$$\alpha f(x) = pdf(X_i) \implies \int \alpha f(x) = \int pdf(X_i) \implies \alpha = \frac{1}{\int f(x)}$$

and thus finding α is equivalent to solving the integral that we are using α to solve.

While the algorithm involves repeatedly sampling some random variable, in practice it is difficult for computers to randomly sample from distributions

that are not simple. In practice, we will use Markov Chain Monte Carlo methods to create a Markov chain with a stationary distribution equal to our chosen pdf, thus eliminating the need for the computer to sample our chosen random variable and instead only requiring it to be able to sample from a uniform random variable to determine if a proposed move should be accepted or rejected.

Using R, I coded a basic integrator that utilizes this method to estimate one-dimensional integrals with which I created the following figures.

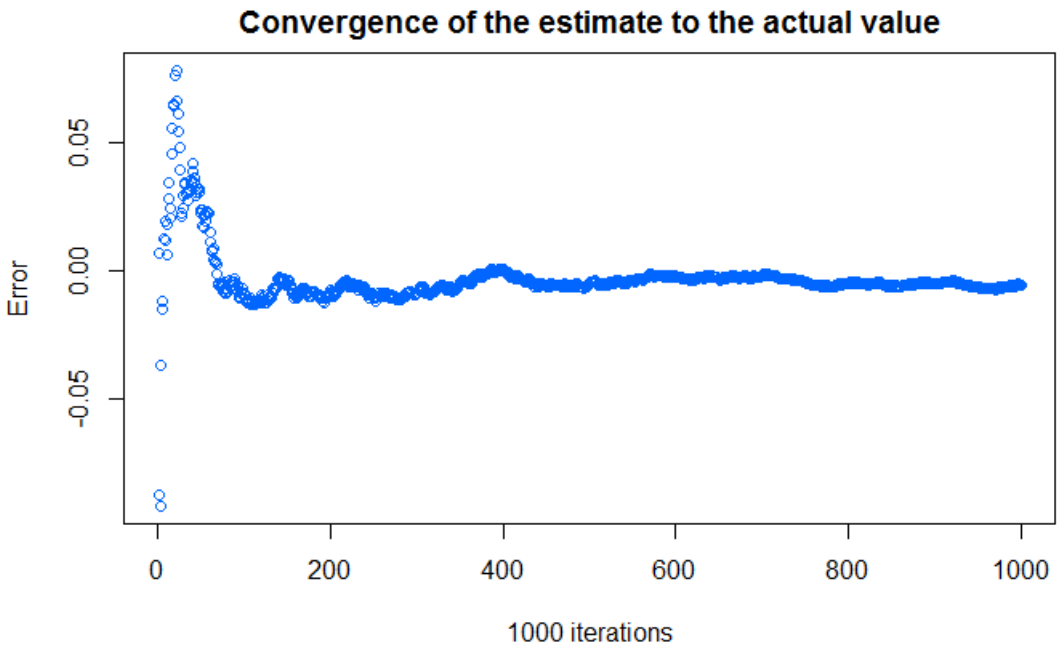

Figure 1: Convergence of the integral on a single run

Figure 1 demonstrates how the estimated value converges to the actual value of the integral over the course of a single run. In this case, the accuracy was checked every 1000 iterations of the Metropolis-Hastings algorithm, and the program ran for a total of 100000 iterations. The y-axis gives the percentage error of the estimate at that point. As we can see, this method can both over- and under-estimate the value of the integral, and as the number of iterations increases we see that the variance of the estimate decreases, as predicted by our derivation of the variance in (4).

To measure the how the error of our estimate compares to the number of

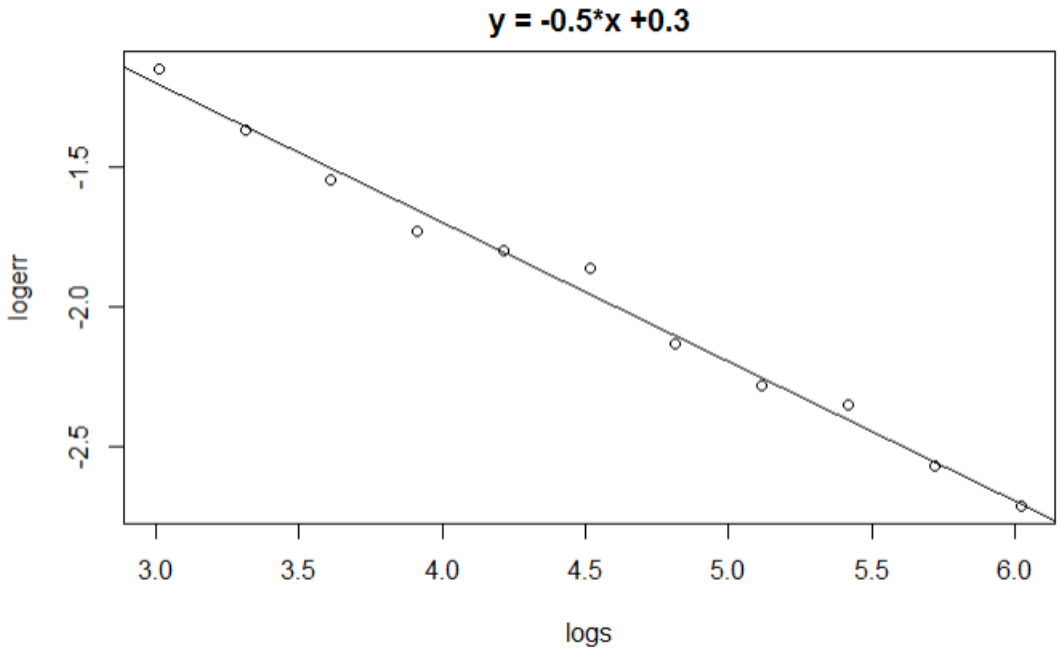

Figure 2: Log-log plot of error vs number of iterations

iterations, we made the following assumption:

$$error = Cn^{\alpha}$$

where C and α are constants. In order to solve for them, we can take the log of both sides to get

$$\log(error) = \log C + \alpha \log n$$

Thus, if we were to make a log-log plot of error vs number of iterations, the slope of the trend line would tell us our value of alpha. To create Figure 2, the program was run twenty times at each of $2^{10}, 2^{11}, ..., 2^{21}$ iterations. Then, the log of the average error for each number of iterations was plotted against the log of the number of iterations. In Figure 2, the trend line has a slope of approximately -0.5, as the line's slope was rounded to four decimal places so we cannot say what it was exactly equal to. Thus, we get that

$$error = Cn^{-0.5} = C\frac{1}{\sqrt{n}}$$

This agrees with the $\frac{1}{\sqrt{N}}$ factor that appeared in our calculation of the standard deviation of our estimate in (4).

5 Appendix

This is the R code used to create the MCMC integrator featured in part 4

```
#Set the upper and lower bounds of integration

lower_bound=1
upper_bound=10

#Calling this function calls the probability density function that has been
chosen for the integration.

probdens=function(x){
dunif(x,min=lower_bound,max=upper_bound)
}

#This function creates a plot of absolute error of the estimated value of
the integral vs its true value

errorplot=function(x,actual){
plot(x,abs(actual-x))
}

#Calling this function calls the function that is to be integrated

intfunc=function(x){
x^2
}

#Calling this function provides a proposal for the next step to move to
in the Markov Chain

proposal=function(a,b){
runif(1,a,b)
}

#Calling this function performs a single iteration of the Metropolis
algorithm

metro=function(x,epsilon=10){
y=runif(1,x-epsilon,x+epsilon)
```

```
if (runif(1)>probdens(y)/probdens(x)){
y=x
}
return(y)
}

#This function performs an estimation of an integral using a Markov
Chain with a transition distribution that the computer draws from directly

mcest=function(n=1000){
acc=rep(0,ceiling(n/100))
states = rep(0,n)
for(i in 1:n){
y = proposal(lower_bound,upper_bound)
states[i]=intfunc(y)/probdens(y)
if(i%%100==0){
acc[ceiling(i/100)]=sum(states)/i
}
}
mylist=list("est"=sum(states)/n,"acc"=acc)
}

#This function performs and estimation of an integral using a Markov
Chain whose distribution is sampled using the Metropolis algorithm. This functio

mcmcest=function(n=1000,epsilon=2){
acc=rep(0,ceiling(n/100))
states = rep(0,n)
y = proposal(lower_bound,upper_bound)
states[1]=intfunc(y)/probdens(y)
numreject=0
for(i in 2:n){
k=metro(y,epsilon)
states[i]=intfunc(k)/probdens(k)
if(y == k) numreject=numreject+1
y=k
if(i%%100==0){
acc[ceiling(i/100)]=sum(states)/i
}
}
mylist=list("est"=sum(states)/n,"acc"=acc,"rejects"=numreject)
```

```
}
```

This is the code used to create the plot measuring convergence of the integral for a single run. For illustration purposes, the code produces several graphs which can be compared to verify that the convergence happens at different rates for different runs of the program.

```
#Set the number of repetitions of the Metropolis algorithm to perform

reps=100000
col=rainbow(5)

#Then, run the program five times and plot the convergence for each run

for(i in 1:5){
obj=mcmcest(reps,5)
plot((obj$acc-integrate(intfunc,lower_bound,upper_bound)$value)/
integrate(intfunc,lower_bound,upper_bound)$value,
ylab="Error",xlab="1000 iterations",col=col[i],
main="Convergence of the estimate to the actual value")
}
```

This is the code used to create the log-log plot for measuring the error/convergence rate of the method when the number of iterations of the Metropolis algorithm is varied.

```
#First, create a table of base 10 logarithms of the number of
iterations performed at each step. Because the algorithm behaves
 poorly forlow valuse of convergence, we will begin at 2^10 iterations

iters=11
logerr=rep(0,iters)
logs=rep(0,iters)
for(i in 1:iters)
{
logs[i]=log10(2^(9+i))
}

#Then, for each step perform the given number of iterations
20 times and take the mean of the base 10 logs of the error
in each attempt
```

```
for(n in 10:(10+iters)){
err=rep(0,20)
for(i in 1:20){
obj=mcmcest(2^n)
err[i]=abs(obj$est-integrate(intfunc,lower_bound,upper_bound)$value)
}
logerr[n-10]=log10(sum(err)/20/integrate(intfunc,lower_bound,upper_bound)$value)
print(n)
}

#Finally, create a plot of the log errors vs log number of
iterations and plot a linear regression to find the slope, which gives
the exponent for our rate of convergence

reg=lm(logerr~logs)
coeff=coefficients(reg)
eq = paste0("y = ", round(coeff[2],4), "*x +", round(coeff[1],1))
plot(logerr~logs,main=eq)
abline(reg)
```