HGEN 48600/STAT 35450: Fundamentals of Computational Biology

Lecture 12 — February 11

Lecturer: Matthew Stephens *Scribe: Katherine Silliman*

Note: These lecture notes are still rough, and have only have been mildly proofread.

Computing practical on implementing MCMC in R. See *github.com/stephens999/mcmc-examples/blob/master/MCMC/IntroMCMC.R* for original code example.

12.1 Ex.1: Sampling from an exponential distribution using MCMC

Any MCMC scheme aims to produce (dependent) samples from a "target" distribution $\pi(x)$. In this case we are going to use the exponential distribution with mean 1 as our target distribution. So we start by defining our target density:

```
target = function(x){
    if(x<0){
        return(0)}
    else {
        return( exp(-x))}
}
```

Recall that the Metropolis-Hastings algorithm is useful for sampling from a distribution that is proportional to our target distribution. It involves the following steps:

1. Initialization: pick an initial state x (usually at random)
2. Randomly draw a state x' from the proposal distribution $Q(x \to x')$
3. Accept the state according to the acceptance distribution H:

$$H = min\left(1, \frac{\pi(x')\,Q(x' \to x)}{\pi(x)\,Q(x \to x')}\right). \tag{12.1}$$

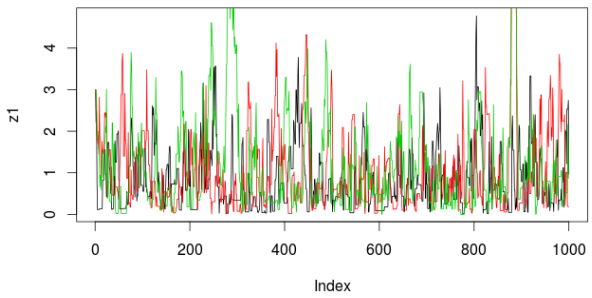
   If not accepted state remains at x, otherwise transitions to x'
4. Save state x, go to #2
5. Repeat desired number of iterations

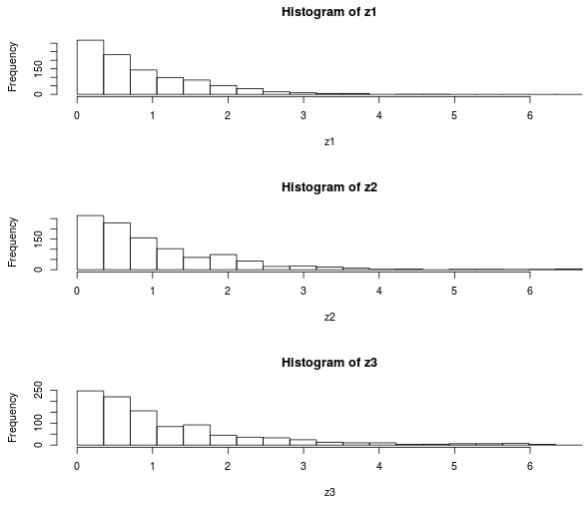
We can code a Metropolis-Hastings scheme in R like this:

```
easyMCMC = function(niter, startval, proposalsd){
    x = rep(0,niter)
    x[1] = startval
    for(i in 2:niter){
        currentx = x[i-1]
        proposedx = rnorm(1,mean=currentx,sd=proposalsd)
        A = target(proposedx)/target(currentx)
        if(runif(1)<A){
            x[i] = proposedx     # accept move with probabily min(1,A)}
        else {
        x[i] = currentx      # otherwise "reject" move, and stay where we are} }
    return(x) }
```

$*$ Note that since $\frac{Q(x' \to x)}{Q(x \to x')} = 1$, we are only computing $\frac{\pi(x')}{\pi(x)}$ to determine acceptance. Plotting x shows us the **trace** of our Markov chain. When x is high dimensional it is more difficult or impossible to view a trace of x.

```
#Run MCMC 3 times and look at how similar the results are.
z1=easyMCMC(1000,3,1)
z2=easyMCMC(1000,3,1)
z3=easyMCMC(1000,3,1)
plot(z1,type="l")
lines(z2,col=2)
lines(z3,col=3)
```





By playing around with the proposal standard deviation, number of iterations, and starting value we can see how these affect the MCMC output.

Starting value: If the starting value is way outside of the target distribution, it will take longer to converge. The code also will not run with a negative starting value.

Proposal SD: A very small proposal SD will make it take longer to converge, whereas a large SD will result in the chain getting "stuck" a lot as values will more frequently get rejected. Intuitively, an SD of 0 will result in a flat trace as x never changes.

Number of iterations: An adequate nuber of iterations are required in order for the chain to converge.

"sticky chain": high autocorrelation among steps
We can also change the target distribution.

```
target = function(x){
    return((x>0 & x <1) + (x>2 & x<3))
}
```

This target will have a bimodal distribution. If the SD is too small (i.e. 0.1) it will not work.

12.1.1 Tuning the MCMC

Tuning your proposal involves finding the best values of the proposal standard deviation in order for the chain to mix faster. This is usually accomplished by ruunning a test chain and looking at the proportion of steps that are accepted. If the proportion of accepted steps is too high, it may slow down mixing. In general, when you have higher dimensions in your target values the optimal optimal acceptance rate is lower.
adaptivity: where you change the rules (i.e. the proposal SD) for an ongoing MCMC chain based on what you see in the trace. This is risky as it may end up not being a true Markov chain. Another important consideration is that the probability of storing an observation must be unbiased and not depend on what stage the chain is currently in. A common way to *thin* values is to set a consistent interval for recorded observations (i.e. every 10 iterations).

12.2 Ex.3: Estimating an allele frequency and inbreeding coefficient (Gibbs Sampler)

In class we glossed over Example 2 in the IntroMCMC.R, which goes over how to implement an MCMC to estimate the allele frequency in a population showing Hardy-Weinberg equilibrium. Example 3 illustrates how to implement a 2 dimensional Markov chain to estimate an

allele frequency and inbreeding coefficient. From IntroMCMC.R: " A slightly more complex alternative than HWE is to assume that there is a tendency for people to mate with others who are slightly more closely-related than "random" (as might happen in a geographically-structured population, for example). This will result in an excess of homozygotes compared with HWE. A simple way to capture this is to introduce an extra parameter, the "inbreeding coefficient" f, and assume that the genotypes AA, Aa and aa have frequencies $fp+(1-f)\,p^2$, $(1-f)\,2p\,(1-p)$, and $f\,(1-p)+(1-f)\,(1-p)^2$ In most cases it would be natural to treat f as a feature of the population, and therefore assume f is constant across loci. For simplicity we will consider just a single locus.

Note that both f and p are constrained to lie between 0 and 1 (inclusive). A simple prior for each of these two parameters is to assume that they are independent, uniform on [0,1]. Suppose that we sample n individuals, and observe nAA with genotype AA, nAa with genotype Aa and naa with genotype aa.

One way we can implement and MCMC routine to get f and p is to update both f and p each iteration and assume that they are independent. Another way is to implement a Gibbs Sampler. To do this, we use a "latent variable" Z_i, a representation of whether an individual came from an inbred mating f or non-inbred mating $(1-f)$. This way we can implement a posterior dstribution for p that does not depend on f.

$$Z_i = \begin{cases} 1, & wpf \\ 0, & wp\,(1-f) \end{cases} \tag{12.2}$$

$Z_i \sim Bernoulli\,(f)$

This makes the likelihoods of the data(genotypes) given p and Z not depend on f:

$$p\,(AA|z_i=1) = p \tag{12.3}$$
$$p\,(AA|z_i=0) = p^2 \tag{12.4}$$
$$p\,(Aa|z_i=1) = 0 \tag{12.5}$$
$$p\,(Aa|z_i=0) = 2p\,(1-p) \tag{12.6}$$
$$p\,(aa|z_i=1) = 1-p \tag{12.7}$$
$$p\,(aa|z_i=0) = (1-p)^2 \tag{12.8}$$

To implement the Gibbs sampler, you would iterate over the following steps:

1. Sample Z from $p\,(z|g,f,p)$
2. Sample f,p from $p\,(f,p|g,z)$