A gentle introduction to MCMC

Arnab Chakraborty

1 Bit of a history

We see the effects of chance everywhere around us–from the roll of a die to the inevitable measurement errors. Many chance phenomena are repetitive in nature. And this provides a way to deal with them. The earliest technique used for this purpose is to wait and see. For example, by repeating a measurement many times we can get an idea about the measurement error. But this may prove too expensive. So the next tenchnique was developed–probability theory, or the mathematics of chance. Long term behaviour could be approximated mathematically using this technique. But unfortunately the math is often too difficult to compute. The advent of modern computers able to generate random numbers revived interest in the old "wait and see" approach. This new version is called the Monte Carlo approach. It is often used even for problems that have no randomness in it.

```
 n= 10000
 x = runif(n, min=-1, max=1)
 y = runif(n, min=-1, max=1)
inside= x*x+y*y<1
 4*mean(inside)
 plot(x,y,col=ifelse(inside,'red','black'),pch=20)
```

This achievement, however, was soon overtaken by ambition. Probability distributions were encountered that were not easy even to simulate from. These distributions came most often from Bayesian computations.

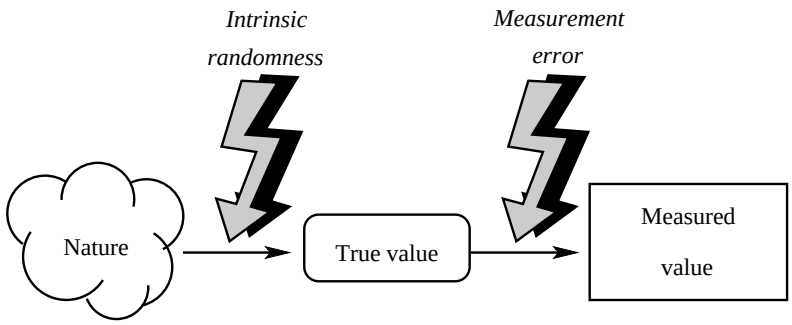
2 A Bayesian excursion

Example: This example is taken from Brandon Kelly's paper [The Astrophysical Journal, 665:1489-1506, 2007] on linear regression with measurement error. Kelly's solution to the problem (which we shall discuss later) is part of the Astrolib library maintained by NASA.

Here we start with the usual linear regression equation

$$Y = a + bX.$$

In a typical astronomical set up, both X and Y are measured quantities. Usually a measurement is subject to two types of randomness–intrinsic randomness affecting the quantity to be measured plus the measurement error. Accordingly, it helps to visualise the entire process as a two-step experiment as shown below.



So behind every measured quantity there is an unobserved true value, which is again random. Let the true values underlying X and Y be ξ and η. Kelly assumes that the intrinsic randomness in ξ can be captured via a Gaussian mixture.

[Eqn 15]

Then η is generated from ξ plus more intrinsic randomness:

[Eqn 13]

Finally he models the measurement errors as

[Eqn 14]

Next we shall specify the priors.

We assume uniform prior for π's.

We assume improper, uniform prior for a, b and σ^2.

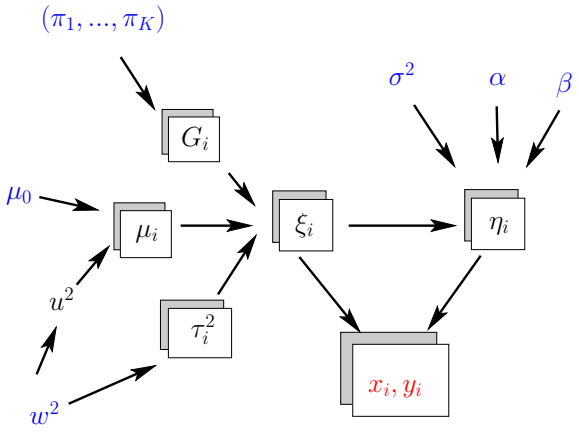
The μ's are given $N(\mu_0, u^2)$ prior and u^2, τ^2's get inverse $w^2\chi^2_{(1)}$-priors.

Finally μ_0 and w^2 are given improper, uniform prior.

[Eqns 43-49]

So the posterior is quite complicated.

The entire set up can be depicted as a *graphical model:*



3 MCMC

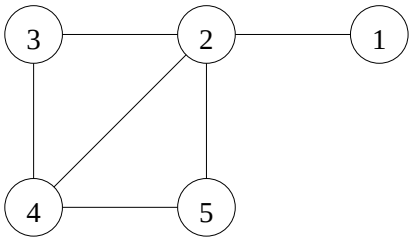
3.1 Basic idea

A novel approach was suggested: MCMC. Consider shuffling a deck of 52 cards. The aim is to select a random permutation out of the 52! permutations. In principle we could do this by–

- drawing one chit at random from 52! chits mixed up in an urn. [Practically impossible]
- picking one cards successively at random, and put them down on the table. [Simpler, but still too cumbersome]

Instead, we simply shuffle the deck –stepwise randomisation. Each shuffle mixes the cards to some extent. Repeated shuffles eventually achieve sufficient randomness. A natural question is: when can a randomisation be split up into steps like this? You can think of this as iterative randomiser, just as the Newton-Raphson method is an iterative equation solver.

Here is another example.



A man starts from vertex 1. At each step he chooses any one of the adjacent vertices randomly, and moves there. Thus his position becomes more and more random. What is the limiting distribution? The answer may be guessed: he would spend more time in vertices with higher number of neighbours. Thus, if N_i denotes the the number of neighbours of vertex i, then the limiting distribution is proportional to

$$(N_1, ..., N_5) = (1, 4, 2, 3, 2).$$

The following simulation confirms this.

```
dest = list(2,c(1,3,4,5),c(2,4),c(2,3,5),c(2,4))

doJump = function(whereNow) {
  if(length(dest[[whereNow]]) == 1)
    dest[[whereNow]]
  else
    sample(dest[[whereNow]],1)
}
```

```
propChain = function(n) {
  x = numeric(n)
  x[1] = 1
  for(i in 2:n)
    x[i] = doJump(x[i-1])

  return(x)
}
```

3.2 Gibbs' sampler

Given a distribution how to come up with a mixing scheme? How to determine when sufficient mixing has been achieved?

A popular method is to use Gibbs' sampler. Consider a man trying to move through a distance that is too long to manage in a single jump. Then he starts walking, which means moving one leg at a time keeping the other foot firmly fixed on ground. Well, that's what a Gibbs' sampler does. To simulate from the joint distribution of $(X_1, ..., X_n)$ it proceeds as follows.

1. First it keeps $X_2, ..., X_n$ fixed, and generates X_1 from the conditional distribution of X_1 given $(X_2, ..., X_n)$.
2. Then it keeps X_1 fixed at this new value, and $X_3, ..., X_n$ at the old values, and generates X_2 from the conditional distribution of X_2 given $(X_1, X_3, ..., X_n)$.
3. The process is continued until we have new values for all the X_i's.
4. The entire process is repeated until convergence.

Why should the full conditionals be easier to simulate from than the joint distribution? Two reasons:

1. One dimensional distributions are easier to simulate from.
2. In a Bayesian set up we can take advantage of conjugate priors.

[Gibb's sampler in Kelly's model: eqns 53-58, 66-68, 73-89, 93-97, 101-103]

What if all the full conditionals are not simple? Use Metropolis-Hastings. We shall discuss it later.

4 Convergence issues

How to determine convergence? No satisfactory answer known. Remember that here we are talking about convergence of distributions, and not the generated numbers. So ideally we should estimate the distribution at different points in the chain, and compare. There are a number of approaches possible:

- Run parallel chains, draw histograms, compare.
- Run one chain, consider well-separated batches, compare using moments like mean or variance.
- Wait for autocorrelation to die down.

4.1 Bottleneck problem

```
condSim = function(given) {
  if(given < 0.9) {
    runif(1,min=0,max=1)
  }
  else if( given < 1) {
    runif(1,min=0,max=given+0.1)
  }
  else if( given < 3) {
    runif(1,min=given-0.1,max=given+0.1)
  }
  else if( given < 3.1) {
    runif(1,min=given-0.1,max=4)
  }
  else {
    runif(1,min=3,max=4)
  }

}


rawSim = function(n = 5000) {
  x = numeric(n)
  y = numeric(n)
  x[1] = 0.5
  y[1] = 0.5
  for(i in 2:n) {
    x[i] = condSim(y[i-1])
    y[i] = condSim(x[i])

  }

  par(mfrow=c(2,2))
  plot(x,xlim=c(0,4),ylim=c(0,4),ty='n',ylab='y',xlab='x')
  polygon(c(0,1,1,3.1,4,4,3,3,0.9,0,0),
          c(0,0,0.9,3,3,4,4,3.1,1,1,0),
          col="pink")
  points(x,y,pch='.')
  plot(y,ty='l',ylim=c(0,4),xlab='step')
```

```
  plot(x,ty='l',ylim=c(0,4),xlab='step')
  par(mfrow=c(1,1))
}


s2 = sqrt(2)
x1 = 1/s2
delX = 0.1/s2
x2 = s2 - delX
ts2 = 3*s2
x3 = ts2 + delX
x4 = ts2 + x1

fs2 = 4*s2

yFromX = function(x) {
  if(x<x1) {
    runif(1,min=-x,max=x)
  }
  else if(x<x2) {
    runif(1,min=x-s2,max=s2-x)
  }
  else if(x<x3) {
    runif(1,min=-delX,max=delX)
  }
  else if(x<x4) {
    runif(1,min=ts2-x,max=x-ts2)
  }
  else {
    runif(1,min=x-fs2,max=fs2-x)
  }
}

y1 = delX
y2 = 1/s2

xFromY = function(y) {
  y= abs(y)

  if(y < y1) {
    runif(1,min=y,max=fs2-y)
  }
  else {
    ifelse(runif(1) < 0.5,
           runif(1,min=y,max=s2-y),
           runif(1,min=ts2+y,max=fs2-y))
```

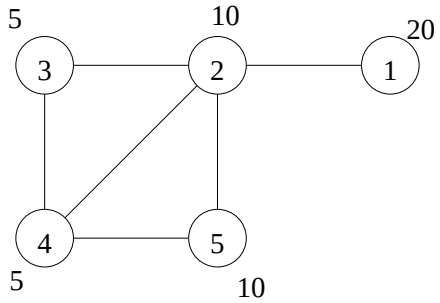
```
  }
}

rotSim = function(n = 5000) {
  x = numeric(n)
  y = numeric(n)
  x[1] = 0.5
  y[1] = 0.5
  for(i in 2:n) {
    x[i] = xFromY(y[i-1])
    if(x[i] >= fs2) stop(paste("Impossible x from y = ",y[i-1]))
    y[i] = yFromX(x[i])
    if(abs(y[i]) >= x1) stop(paste("Impossible y from x = ",x[i]))
  }

  par(mfrow=c(2,2))
  plot(x,xlim=c(0,fs2),ylim=c(-x1,x1),ty='n',xlab='x',ylab='y')
  polygon(c(0,x1,x2,x3,x4,fs2,x4,x3,x2,x1,0),
          c(0,-x1,-y1,-y1,-x1,0,x1,y1,y1,x1,0),
          col="pink")
  points(x,y,pch='.')
  plot(y,ty='l',ylim=c(-x1,x1),xlab='step')
  plot(x,ty='l',ylim=c(0,fs2),xlab='step')
  par(mfrow=c(1,1))
}
```

5 Metropolis-Hastings' method

If Gibbs' sampler is compared to walking, then Metropolis-Hastings' method is a generalisation that also allows jumps with both feet off the ground simultaneously. You do not *have* to jump, walking is still allowed. So Gibbs' sampler is a special case of Metropolis-Hastings' method.

Metropolis-Hastings' method has a surprising amount of generality. It starts with any Markov Chain (ie, any set of jump rules), and tweaks it so that it converges to any target distribution of our choice! Well, that's bit of an overstatement. We do need the initial chain to satisfy *some* conditions, but they are really weak conditions.

Consider the graph walking example again. We started with the Markov Chain (jump rule): jump to any adjacent vertex with equal probability. This causes us to visit vertices with more neighbours to be visited more often. But suppose that we want to visit them randomly according to the probabilities proportional to the following labels:



Then Metropolis-Hastings' method modifies the original Markov Chain as follows. Suppose we are at 2, and we plan to jump to 3. But we do not immediately carry out our plan. Instead, we check the two labels, π_2 and π_3. We jump with probability

$$\min\left\{1, \frac{\pi(3)p(2|3)}{\pi(2)p(3|2)}\right\}.$$

Considering the case $p(2|3) = p(3|2)$, we can say that we are more willing to jump to higher labels than to lower ones.

```
MHchain = function(prob=rep(1,5),n=5000) {
  x = numeric(n)
  x[1] = 1
  for(i in 2:n) {
    proposal = doJump(x[i-1])
    p = (prob[proposal]*length(dest[[x[i-1]]]))/(prob[x[i-1]]*length(dest[[proposal]]))
    if(p >= 1) {
      x[i] = proposal
    }
    else {
      if(runif(1)<p)
        x[i] = proposal
      else
        x[i] = x[i-1]
    }
  }
  return(x)
}

display = function(x) {
  n = length(x)
  par(mfrow=c(1,1))
  plot(0,xlim=c(0,4),ylim=c(0,3),ty='n',xlab='',ylab='',bty='n',xaxt='n',yaxt='n')
  px = c(3,2,1,1,2)
  py = c(2,2,2,1,1)
  lines(px,py)
  lines(px[c(5,2,4)],py[c(5,2,4)])
```

```
  points(px,py,cex=10*table(x)/n,col="red",lwd=3)
}
```