A detailed reference of MCMC algorithms

Siddharth Bhat (20161105) `siddu.druid@gmail.com`

April 13, 2020

1 Why do we need MCMC? A practitioner's perspective

Consider that we are plonked down in the `C` programming language, and our only method to generate random numbers is to call `int rand(void)`. However, *the type is a lie*, since we are able to modify global mutable state. So, really, to have a mathematical discussion about the whole state of affairs, we will write the type of `rand` as $\texttt{rand} : S \to S \times \texttt{int}$ — that is, it receives the entire state of the program, and then returns the `int`, along with the new state of the program. $\texttt{uniformbool} : \texttt{S} \to \texttt{S} \times \{0, 1\}$.

When we say that `rand` generates random numbers, we need to be a little more specific: what does it mean to generate random numbers? we need to describe the *distribution* according to which we are receiving the random numbers from the random number generator *rand*. What does that mean? Well, it means that as we generate more numbers, the *empirical distribution* of the list of numbers we get from the successive calls to `uniformbool` tends to some *true distribution*. We will call this *true distribution* succincty as **the** distribution of the random number generator. Formally, let us define $F(t) \equiv \int_0^t P(x)dx$ to be the cumulative distribution of P.

In the case of `uniformbool`, we are receiving random numbers according to the distribution:

$$P_{\texttt{uniformbool}} : \{0, 1\} \to [0, 1]; \qquad P_{\texttt{uniformbool}}(x) = 1/2$$

That is, both 0 and 1 are *equally likely*. However, this is extremely boring. What we are *usually* interested in is to sample $\{0, 1\}$ in some *biased* fashion:

$$P^{bias}_{\texttt{uniformbool}} : \{0, 1\} \to [0, 1]; \qquad P^{bias}_{\texttt{uniformbool}}(0) = bias; \qquad P^{bias}_{\texttt{uniformbool}}(1) = 1 - bias$$

And far more generally, we want to sample from *arbitrary domains* with *arbitrary distributions*:

$$\texttt{sampler}^P_X : S \to S \times X;$$

This function is boring. What we *really* want to sample from are more interesting distributions. For example:

- The normal distribution $P(x : \mathbb{R}) = e^{-x^2}$.
- The poisson distribution $P(x : \mathbb{N}) = e^{-\lambda}\lambda^n/n!$.
- A custom distribution $P(x : \mathbb{R}) = |sin(x)|$.

1.1 Fundamental Problem of MCMC sampling

Given a weak, simple sampler of the form $\texttt{rand} : S \to S \times \texttt{int}$, build a sampler $\texttt{sampler}(P, X) : T \to T \times X$ which returns value distributed according to some distribution of choice $P : X \to [0, 1]$.

1.2 Sampling use case 1. Simulation And Sampling

1.3 Sampling use case 2. Gradient Free Optimisation

1.4 Sampling use case 3. Numerical Integration

2 Where it all begins: The Metropolis Hastings sampler

3 Gibbs sampling

We wish to sample from a joint distribution $P(X_1, X_2, X_3)$. However, it might be far cheaper to sample $P(X_1|X_2, X_3)$, $P(X_2|X_1, X_3)$, and $P(X_3|X_1, X_2)$. If it is indeed cheaper, then we can use a Gibbs sampler to draw from the actual distribution $P(X_1, X_2, X_3)$ by combining samples from the *conditional* distribution cleverly. The code is:

```
# sampler_x: y, z -> new x
# sampler_y: x, z -> new y
# sampler_z: x, y -> new z
# N: number of iterations.
# returns: new (x, y, z)
def sampler_xyz(sampler_x, sampler_y, sampler_z, x0, y0, z0, N):
    (x, y, z) = (x0, y0, z0)
    for i in range(N):
        x = sampler_x(y, z)
        y = sampler_y(x, z) # NOTE: use *new* x
        z = sampler_z(x, y) # NOTE: use *new* x, y
    return (x, y, z)
```

4 Hamiltonian Monte Carlo

If our target probability density $P : X \to [0, 1]$ is *differentiable*, then we can use the derivative of $P(x)$ to provide better proposals. The idea is as follows:

- Interpret the probability landscape as actual terrain, with points of high probability being "valleys" and points of low probability being "peaks" (ie, invert the probability density with a transform such as $terrian(x) = e^{-P(x)}$
- For a proposal at a position $x_0 \in X$, keep a ball at x_0, *randomly choose its velocity*, simulate the ball according to classical mechanics (Newton's laws of motion) for a fixed duration $D \in \mathbb{R}$ and propose the final position as the final position of the ball. This is reversible and detail balanced because *classical mechanics is reversible and detail balanced.*

We need the derivative of $P(x)$ to be able to simulate the ball according to Newton's laws. If we can do this, though, we are able to cover large amounts of terrain.

4.1 Newton's laws of motion: Hamilton's equations

$$\begin{aligned}\frac{\partial \mathbf{q}}{\partial t} &= \frac{\partial H}{\partial \mathbf{p}} \\ \frac{\partial \mathbf{p}}{\partial t} &= -\frac{\partial H}{\partial \mathbf{q}}\end{aligned}$$

4.2 Detail balance

4.3 Hamiltonian for our simulation

If we have a a target probability distribution $Prob(\mathbf{q}) : \mathbb{R}^n \to \mathbb{R}$ we choose the Hamiltonian to be $H(\mathbf{p}, \mathbf{q}) \equiv Prob(\mathbf{q}) + \mathbf{p}^T\mathbf{p}/2$.

4.4 Simulation: Naive

4.5 Simulation: Need for symplectic integrators

4.6 Simulation: Final

We no longer need to choose how many steps to walk with a no-U-turn-sampler. It prevents us from re-walking energy orbits, by detecting when we have completed traversing an orbit and are going to take a "U-turn". The details are quite complex, so we may not cover this here.

5 Discontinuous Hamiltonian monte carlo

What if our distribution P is *discontinuouse*? How do we perform MCMC in that case?

6 Low discrepancy sequences

Low discrepancy sequences are sequences of numbers that more evenly distributed than pseudorandom numbers in high dimensional space. Hence, simulations which use Low discrepancy sequences generally approximate integrals faster than psuedo-randomly generated points.

Formally, let us consider the S dimensional half-open cube $\mathbb{I}^S \equiv [0,1)^S$. assume we have a set of points $P \subseteq \mathbb{I}^S$, and a sub-interval $B \subseteq I^S$, where a sub-interval is a subset of the form $B \equiv \prod_{i=1}^{S} \{x \in \mathbb{I} : a_i \leq x \leq b_i\}$.

Given a universe set X and two subsets $Large, Small \subseteq X$, we define the amount of containment of $Small$ in $Large$ to be $C(Small, Large) \equiv \frac{|Large \cap Small|}{|Large|}$.

Intuitively, this measures the fraction of $Small$ that is in $Large$. We now define the discrepancy of the set of points P relative to the sub-interval B as:

$$\begin{aligned}
&D(B,P) \equiv \left|C(P,B) - C(B,\mathbb{I}^S)\right| \\
&= \left|\frac{|B \cap P|}{|P|} - \frac{|B \cap \mathbb{I}^S|}{|\mathbb{I}^S|}\right| \\
&= \left|\frac{|B \cap P|}{|P|} - \frac{Volume(B)}{1}\right| \\
&= \left|\frac{|B \cap P|}{|P|} - Volume(B)\right|
\end{aligned}$$

So, the discrepancy is measuring if P fits within B the way B fits within the full space.

Now, the **worst-case-discrepancy** is defined as the maximum discrepancy over all sub-intervals:

$$D^{\star}(P) \equiv \max_{B \in \mathcal{J}} D(B,P)$$

where $\mathcal{J} \subseteq 2^{\mathbb{I}^S}$ is the set of all sub-intervals:

$$\mathcal{J} \equiv \{\{x \in \mathbb{I}^S : l[i] \leq x[i] \leq r[i] \ \forall i\} : \vec{l}, \vec{r} \in \mathbb{I}^S\}$$

The goal is a *low discrepancy sequence* is to minimise the worst-case-discrepancy.

6.1 Error bounds for Numerical integration

6.2 Sobol sequences

Sobol sequences are an example of low-discrepancy sequences

7 Future Work!

I've left the topic names up because they have interesting ideas, which I have sketched out in brief. Writing them down formally will take far too long; Hence, they've been left here [for the purposes of the project]. I will probably update this document even after the project, once I have grokked this material better.

8 Slice Sampling

9 No-U-Turn sampling