http://www.math.nyu.edu/faculty/goodman/teaching/MonteCarlo17/

Class notes: Monte Carlo methods
Part 1, Introduction

Jonathan Goodman

January 20, 2017

1 Introduction to Monte Carlo

Monte Carlo methods are computational methods that use random numbers. Randomness may be part of the original problem, or it may not. Finding the expected value of a random variable is an example where randomness is part of the problem. Suppose $X \in \mathbb{R}^d$ is a random variable with probability density $f(x)$, and we want an expectation value

$$A = \mathrm{E}_f[V(X)] = \int V(x) f(x)\, dx \;. \tag{1}$$

Even in this case the answer, A, is not itself a random number. It may or may not be practical to calculate or estimate A without Monte Carlo.

Monte Carlo also can solve some problems that do not, at least not initially, involve randomness. For example, in some machine learning situations there are data examples Y_k for k up to $N = 10^8$ (say). The problem is to "train" an algorithm which has a parameter set $x = (x_1, \ldots, x_d) \in \mathbb{R}^d$. This may mean minimizing an "objective function"

$$F(x) = \sum_{k=1}^{N} f(x, Y_k) \;.$$

Using gradient descent with "learning rate"[1] α, the parameter optimization process may produce

$$x^{(i+1)} = x^{(i)} - \alpha \nabla F(x^{(i)}) \;.$$

The problem is that N may be so large that ∇F expensive to calculate. Using Monte Carlo, one may select $M \ll N$ data values "at random" and estimate the gradient using

$$\nabla F(x) \approx \frac{N}{M} \sum_{j=1}^{M} \nabla f(x, Y_{k_j}) \;.$$

Procedures like this are called *stochastic gradient descent.* Here, the only reason not to use all the data at every iteration is that it is too expensive.

These notes, at least the first six "weeks" of them, are aimed at a very diverse audience. One often talks about the "community" or people doing X, as we just did for machine learning. It is fair to say that people doing Monte

[1] The machine learning community has a habit of renaming things, so "calibration" becomes "train", "step size" becomes "learning rate", etc.

Carlo do not form a community in this sense. Part of the reason is that Monte Carlo methods are application specific, so it is hard for someone using Monte Carlo for engineering risk analysis (say) to interact very deeply with someone doing stochastic gradient descent, or someone estimating the entropy of large molecules, or someone using Bayesian statistics in econometrics. The Monte Carlo class at the Courant Institute of NYU also has a very diverse audience. I will try to address as wide a group as possible. The early topics are basic to most Monte Carlo applications.

One of the basic tasks in Monte Carlo is *sampling.* You have a probability density or distribution, f, and you want random samples $X \sim f$. At first, we cover *direct* sampling methods, which are methods that make independent samples. Then we will spend several weeks covering *Markov chain Monte Carlo*, or *MCMC*. These have increased the range of probability distributions that have practical samplers. MCMC is a very active are of research. People are discovering surprising tricks that allow better sampling and open up new areas of application. Much MCMC research is practical – find methods – and much is theoretical – analyze methods. Analysis of MCMC is beautiful mathematics and has been helpful for practice, but not as much as other parts of computational science and engineering. Hopefully some people in these Monte Carlo classes can fix that.

Many Monte Carlo problems seem to have versions of the *rare event* difficulty. A simple version of this problem is that there is a set R that is "rare" in the sense that[2]

$$\Pr(R) = \int_R f(x)\, dx = \varepsilon \ll 1 \;,$$

but which determines "most" of the integral in the sense that

$$A_R = \int_R V(x) f(x)\, dx \approx A \;.$$

This happens when $V(x)$ is very small when $x \notin R$. This happens, for example, in some Bayesian statistics applications. There, $V(x)$ is a measure of how well a "random" parameter set x fits the data. It could happen, if you are lucky, that only very specific parameter combinations fit the data well, while random parameter sets are mostly poor fits. Direct methods that rely on samples of f often have trouble with problems like these because most of the samples don't contribute much to the answer. These notes will cover some traditional approaches, including importance sampling, and some alternatives that go by names such as *thermodynamic integration* and *nested sampling.* These methods are similar, but have different names because there is not a coherent Monte Carlo community where people trade methods.

Error estimation is an important, difficult, frustrating, and often neglected part of Monte Carlo practice. Professionals using Monte Carlo (as students in Monte Carlo class aspire to become) make, or should make, estimates of the accuracy of a result found by Monte Carlo. These range from simple error

[2] Mathematicians call a quantity ε to indicate that it is small.

bars in direct sampling Monte Carlo to *auto-correlation* analysis for MCMC, to recursive variance estimates in nested sampling. Writing from experience, I can say that this is one of the ways the Monte Carlo expert contributes to a multi-disciplinary research team.

This course is aimed at someone who will do Monte Carlo in some application. This means programming. Monte Carlo codes typically are expensive and slow to run, so computer *performance* (how fast a program executes an algorithm vs. how fast a different program in a different language would execute the same algorithm) is a major concern. Scientific scripting languages (Python, R, Matlab) have highly optimized vector operations, so they perform well on large scale structured linear algebra problems. There may be little benefit to using a "faster" complied language such as C++ or Fortran. Monte Carlo algorithms, particularly those involving MCMC, may have worse performance when coded in Python or Matlab or R than in C++. But this class will not emphasize performance issues. The examples in the text will be in Python 2.7.

2 Direct sampling methods

It is a sad fact that math classes start with the most gritty technical part of their subject. Real analysis starts with set theory and sigma algebras. Numerical computing starts with the IEEE floating point standard. In that spirit, this Monte Carlo course starts with the direct sampling methods that are at the deepest level of most Monte Carlo codes.

Suppose $f(x)$ is the probability density for an n component random variable X. A *direct sampler* is an algorithm or a piece of software that produces independent random variables with the density f. This is written $X \sim f$. Consider the code fragment

```
X1 = fSamp()
X2 = fSamp()
```

If `fSamp()` is a direct sampler of f, then $X_1 \sim f$, and $X_2 \sim f$, and X_1 is independent of X_2.

Many probability distributions you meet in practice do not have practical direct samplers. Sampling them requires MCMC. But direct samplers are important parts of most MCMC methods.

3 Pseudo random number generators

A *pseudo random number generator* is the basic ingredient of any sampler. A perfect random number generator (we usually drop the "pseudo") would be a procedure `uSamp()` so that `U[i] = uSamp();` (in a loop over `i`) would fill the array `U` with independent random variables uniformly distributed in the interval $[0, 1]$. The word "pseudo" tells us that the numbers are not actually random, but are produced by a deterministic algorithm. Modern state-of-the-art random number generators produce “random” numbers that good enough for any Monte Carlo computation I am aware of.

Random number generators in common use have the a common structure. There is a *state*, s, that is some small amount of data. *Congruential* random number generators have a state that is an integer $s \in \{0, 1, \ldots, p-1\}$, where p is a large prime number. More sophisticated generators have a state that may consist of several integers or other discrete data. A random number generator is defined by two functions, a state update function Φ, and an output function Ψ. The state update function produces a new state from the current one. If the current state is s_n, the next state is $s_{n+1} = \Phi(s_n)$. The output function produces a number in the interval $[0, 1]$ from the state. If s_n is the current state, then $U_n = \Phi(s_n)$ is the corresponding “random” number. A call `U = uSamp()` has two effects. It returns the number $U = \Psi(s)$, and it updates the state: $s \leftarrow \Phi(s)$. If you start with an s_0 and call `uSamp()` many times in succession, you will get the sequence $U_n = \Psi(s_n)$, where $s_{n+1} = \Phi(s_n)$. If you start again with the same s_0, you will get the same sequence U_n.

Download the file `RandomNumberGenerator.py` from the class web site (URL at the top of the page). The random number library is called `random`. It is imported as `rn` in all examples in this class. It produces five sets of three random numbers uniformly distributed in $[0, 1]$. The basic function that updates the seed and outputs a number is called `random()`. The procedure `getstate()` returns the state but doesn’t change the state or effect the sequence being generated. The procedure `seed(`*`seed`*`)` sets the state using some function of the seed given. The seed typically will be an integer, `17` in this case. You use a seed at the beginning of a Monte Carlo run so that the output is the same every time. Python gets the initial seed in some way from the system clock, so if you don’t set the initial state using a seed or in some other way, the results will be different every time. If you run `RandomNumberGenerator.py` again, you will see that the first set of numbers is different. But the third set should be the same, because you set the state from the same seed (`17`) each time. That’s why the third and fourth sets are identical. The fifth set is generated from the same starting state as the second set, so it is identical to the second set.

4 Mapping methods for direct sampling

I.i.d uniformly distributed random variables are used to generate all the other random variables in Monte Carlo. We usually assume that the uniform random number generator is perfect, that it produces a sequence U_n that is exactly i.i.d. and uniformly distributed in $[0, 1]$. A *sampler* is an algorithm that uses one or more such uniforms to generate samples from other densities.

A *direct sampler* of density f uses one or more uniforms to generate an independent sample $X \sim f$. Suppose it takes k uniforms to generate X. Mathematically, this means that there is a function $x = G(u_1, \ldots, u_k)$ so that if the U_j are i.i.d. uniforms, and $X = G(U_1, \ldots, U_k)$, then $X \sim f$. This section gives some examples of direct samplers where k is known in advance. These are *map-*

ping methods, with G being the mapping. The next section discusses rejection methods, where k is not known in advance.

4.1 Exponential random variables

The *exponential* distribution with rate parameter λ has probability density $f(t) = \lambda e^{-\lambda t}$, if $t \geq 0$, and $f(t) = 0$ if $t < 0$. You can think of an exponential random variable, T, as the time you have to wait until "a bell rings". The restriction $T \geq 0$ is natural if $t = 0$ represents the present. The probability that the bell rings right away is $P(0 \leq T \leq dt) = f(0)\,dt = \lambda\,dt$. We call λ the *rate constant* because it is the "rate" of bell ringing at the beginning. The exponential random variable is special in that it has no memory. If the bell has not rung by time s, it is as though time starts over then. More precisely: $P(T \in [s, s+dt] \mid T \geq s) = \lambda\,dt$. The conditional probability density is $f(t \mid T \geq s) = e^{-\lambda(t-s)}$. (The reader should do the easy verification that the two conditional statements are equivalent.)

Exponentials (i.i.d. exponential random variables with arbitrary λ) have many uses in Monte Carlo. The occupation times of a continuous time Markov chain are exponential. The Green's function commonly used in *Green's function Monte Carlo*, or *GFMC*, is sampled using an exponential, see Exercise 4.

To make an exponential, just set

$$T = \frac{-1}{\lambda} \log(U) \ , \qquad (2)$$

where U is uniform $[0, 1]$. We verify this by computing the probability density of the random variable T defined by (2). This density is defied by

$$f(t)\,dt = P(T \in [t, t + dt]) \ .$$

But we can calculate that this event says about U:

$$\begin{aligned} t \leq T \leq t + dt &\iff t \leq \frac{-1}{\lambda}\log(U) \leq t + dt \\ &\iff -\lambda t - \lambda dt \leq \log(U) \leq -\lambda t \\ &\iff e^{-\lambda t} e^{-\lambda dt} \leq U \leq e^{-\lambda t} \\ &\iff e^{-\lambda t}(1 - \lambda dt) \leq U \leq e^{-\lambda t} \ . \end{aligned}$$

This is an interval in $[0, 1]$ of length $e^{-\lambda t}\lambda dt$. Since U is uniformly distributed in $[0, 1]$, the probability of this is $e^{-\lambda t}\lambda dt$. This implies that $f(t) = \lambda e^{-\lambda t}$. A careful reader may worry that we implicitly assumed that $t > 0$. The formula (2) does not produce negative T if $U \in [0, 1]$.

Here are two checks of (2). Since $U \leq 1$, we have $T > 0$. That is why we need the minus sign. If the random number generator gives $U = 0$, then T is not defined. Unfortunately, some random number generators do that from time to time, so you might need to check that $U \neq 0$ in the code before applying (2). The other check involves the λ factor. If λ is large, the rate is fast and T happens sooner. Our formula does that. To say this in a deeper way, the units of λ are 1/Time because λ is a rate. Therefore $1/\lambda$ has the same units as T.

4.2 Inverting the CDF

If X is a one dimensional random variable, the *cumulative distribution function*, or *CDF*, is $F(x) = P(X \leq x)$. For example, a standard uniform CDF is $F(u) = u$ for $u \in [0,1]$, $F(u) = 0$ for $u < 0$, and $F(u) = 1$ for $u > 1$. An exponential with rate constant λ has CDF $F(t) = 1 - e^{-\lambda t}$ for $t \geq 0$, and $F(t) = 0$ for $t < 0$. In general, the CDF of X is an increasing function of x, and it is strictly increasing for any x with $f(x) > 0$. Also, $F(x) \to 0$ as $x \to -\infty$, and $F(x) \to 1$ as $x \to \infty$.

If X is a random variable with $F(x)$ as its CDF, then the random variable $U = F(X)$ is uniform in $[0,1]$. Conversely, if U is uniform $[0,1]$, and we find X by solving the equation $F(X) = U$, then $X \sim f(x) = F'(x)$. This is "obvious". If $u \in [0,1]$, and x is some number so that $u = F(x)$, then the events $U \leq u$ and $X \leq x$ are the same, so they have the same probability. The probability that $U < u$ is $u = F(x)$. Therefore $F(x)$ is the probability that $X < x$. You have to be careful about degenerate cases where $f(x)$ vanishes (e.g., the exponential for $t < 0$) and partly discrete random variables where $F(x)$ can be discontinuous. Otherwise, there is a unique x for each $u \in [0,1]$ and a unique u for each x, and the inverse function $x = F^{-1}(u)$ is well defined.

$$X = F^{-1}(U) \tag{3}$$

generates a sample $X \sim f$.

For example, you can generate an exponential with rate constant λ by solving $F(T) = U$, which given $1 - e^{-\lambda T} = U$ and then

$$T = \frac{-1}{\lambda} \log(1 - U) \ .$$

This is the same as (2) because $1 - U$ has the same uniform distribution as U. Another example, with $f(r) = Cr^n$, for $0 \leq r \leq 1$ and $f = 0$ otherwise, is in Exercise 3.

It is not quite so simple for normals. The CDF, written $N(x)$, is not an elementary function. Nevertheless, there is fast and accurate software that computes $N(x)$ and $N^{-1}(u)$ quickly and almost to machine precision. There is a separate method for generating normals, the Box Muller algorithm (described below). This is very elegant, but the Python, R, and Matlab normal random number generators use the inverse of $N(x)$, not Box Muller.

4.3 Coin tossing

Suppose you want a discrete random variable $X = 1$ with probability p and $X = 0$ with probability $q = 1 - p$. You just generate a uniform, U, and say $X = 1$ if $U < p$, and $X = 0$ otherwise.

```
X = 0
if  (rn.random())< p:
     X = 1
```

More generally, suppose you want $X = k$ with probability p_k, and $p_1 + \cdots + p_n = 1$. The discrete distribution function is

$$P_k = \sum_{j \leq k} p_j \ .$$

The following code produces the desired sample

```
X = 0
U = np.random()
while (P[X] < U):
    X = X + 1
```

This code assumes that $P_n = 1$ exactly. You can put this in the code with

```
P[n] = 1.;
```

In IEEE floating point arithmetic, this represents the mathematical 1 exactly. If you get P_n by adding the p_j, roundoff error could give a result slightly smaller than the mathematical 1. You would have to hope that roundoff error in the random number generator never produces $U > 1$. See the code `DiscretRandomVariables.py`.

4.4 Normals, Box Muller

The *standard normal* probability density is

$$f(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2} \ .$$

The Box Muller algorithm is a mapping that turns two independent uniforms into two independent standard normals. The algorithm is elegant and easy to program. It is based on the trick for computing the Gaussian integral

$$I = \int_{-\infty}^{\infty} e^{-x^2/2} \, dx \ .$$

The trick is to write

$$\begin{aligned}
I^2 &= I \cdot I \\
&= \int_{-\infty}^{\infty} e^{-x^2/2} \, dx \cdot \int_{-\infty}^{\infty} e^{-y^2/2} \, dy \\
&= \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} e^{-x^2/2} e^{-y^2/2} \, dxdy \\
&= \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} e^{-(x^2+y^2)/2} \, dxdy \ .
\end{aligned}$$

Then switch to polar coordinates $x = r\cos(\theta)$, $y = r\sin(\theta)$, $dxdy = d\theta\, rdr$, and $x^2+y^2 = r^2$. Therefore

$$\begin{aligned} I^2 &= \int_{r=0}^{\infty}\int_{\theta=0}^{2\pi} e^{-r^2/2}\, d\theta\, rdr \\ &= 2\pi \int_{r=0}^{\infty} e^{-r^2/2}\, rdr \end{aligned}$$

To work the last integral, use $s = r^2/2$, $ds = rdr$, which gives

$$I^2 = 2\pi \int_0^\infty e^{-s}\, ds = 2\pi \ .$$

There is no explicit formula for the one dimensional indefinite integral $\int e^{-x^2/2} dx$. But the indefinite integral in two dimensions is $\int e^{-r^2/2}\, rdr$. We find this indefinite integral using the formula $\frac{d}{dr}e^{-r^2/2} = -re^{-r_2/2}$.

Here is the Monte Carlo version of this trick. We seek to make a pair, (X,Y), of independent standard normals. The probability density is $\frac{1}{2\pi}e^{-(x^2+y^2)/2}$, which is isotropic. See Exercise 1 for a related consequence of the fact that the joint density of independent standard normals is isotropic. The polar coordinate representation of $(X,Y) = (R\cos(\Theta), R\sin(\Theta))$ involves a random distance, R, and a random angle Θ. Clearly, Θ is uniformly distributed in $[0,2\pi]$, so we can generate it using

$$\Theta = 2\pi U \ , \tag{4}$$

where U is uniform $[0,1]$. Clearly R and Θ are independent. The density of R is $f(r) = re^{-r^2}$. This is because

$$f(r)\, dr = P(r \leq R \leq r{+}dr) = \frac{1}{2\pi}\int\int_{r\leq R\leq r+dr} e^{-(x^2+y^2)/2}\, dxdy = e^{-r^2/2}\, rdr \ .$$

Exercise 5 uses the method of Subsection 4.2 to show that a sampler for this distribution is

$$R = \sqrt{-2\log(V)} \ , \tag{5}$$

where V is uniform in $[0,1]$. It is possible to find an explicit sampler because there is an explicit formula for the indefinite integral of f. Using the Box Muller algorithm you can fill an array Z with n i.i.d. standard normals

```
i = 0
for j in range(0,n_by_2):
    Th = 2*np.pi*U[j]
    R  = np.sqrt( -2*np.log(V[j]))
    X[i] = R*np.cos(Th)
    i    = i+1
    X[i] = R*np.sin(Th)
    i    = i+1
```

The code BoxMuller.py does this.

4.5 Order statistics

You may never use this clever trick, but it does illustrate the possibility of using several uniforms to generate a single $X \sim f$. Suppose U_1 and U_2 are independent standard uniforms. An elementary verify shows that $X = \max(U_1, U_2)$ has density $f(x) = 2x$, if $0 \leq x \leq 1$, and $f(x) = 0$ otherwise. Going further, the result of sorting $(U_1, \ldots, U_n)$ is written $U_{(1)} \leq \cdots \leq U_{(n)}$. The k^{th} smallest of $(U_1, \ldots, U_n)$ is the k^{th} *order statistic*, $U_{(k)}$. If $n = 2$, then $X = U_{(2)} = \max(U_1, U_2)$ as above. If $n = 3$, the density of $X = U_{(2)}$ is $6x(1-x)$. The density goes to zero as $x \to 0$ or $x \to 1$ because it is hard for the middle of three to be very close to zero or 1. The density of $U_{(k)}$ has a factor of x for each $j < k$ and a factor of $(1-x)$ for each $j > k$. These densities do not arise so often by themselves, but they can be useful as proposals for rejection sampling discussed in Section 6.

5 Multivariate normal sampling via Cholesky factorization

High dimensional distributions rarely have practical direct samplers. The multivariate normal is the crucial exception. A multivariate normal with mean μ and covariance matrix C, or *precision* matrix $H = C^{-1}$ has probability density

$$f(x) = \frac{1}{(2\pi)^{n/2} \det(C)^{1/2}} e^{-(x-\mu)^t H (x-\mu)/2} \,. \qquad (6)$$

If X has this density, we write $X \sim \mathcal{N}(\mu, C)$. In one dimension the covariance matrix is just the variance σ^2, where σ is the standard deviation. If Y is another multivariate normal, we write μ_X and μ_Y, C_X and C_Y for the parameters of the distributions.

Suppose $X \sim \mathcal{N}(\mu_X, C_X)$. Let $Y = AX + b$, where A is an $m \times n$ matrix with rank m. This requires $m \leq n$. Then Y is multivariate normal with parameters $\mu_Y = A\mu_X + b$ and $C_Y = AC_X A^t$. You can derive the covariance formula (if $\mu_X = 0$, $b = 0$, and $\mu_Y = 0$) using $C_Y = E[YY^t] = E[(AX)(AX)^t] = E[A(XX^t)A^t] = AE[XX^t]A^t = AC_X A^t$.

It is easy to generate a multivariate normal $Z \sim \mathcal{N}(0, I)$, just take the components of Z to be independent standard normals. If we take $X = AZ + b$, we get $X \sim \mathcal{N}(b, AA^t)$. This gives $X \sim \mathcal{N}(\mu_X, C_X)$ if $b = \mu_X$ and $C_X = AA^t$. If C_X is known, we can find a suitable matrix A using, for example, the Cholesky factorization. If C is a symmetric positive definite matrix (as any covariance matrix must be), there is a *Cholesky factor* L that is lower triangular and has $C = LL^t$. There is good software in most programming languages to compute Cholesky factors. In C, C++, or Fortran, you can use LAPACK. In Matlab or Python you can use built in functions. The drawback is that Cholesky factors are somewhat expensive to compute in high dimensions. But $n = 1000$ is still practical.

It often happens that you have $H = C^{-1}$ rather than C, an example is in Subsection 6.2. It may be that H is tridiagonal or for another reason has a simple Cholesky decomposition. If $H = MM^t$ is the Cholesky factorization of H, then $H^{-1} = \left(M^t\right)^{-1} M^{-1}$. Recall that $\left(M^t\right)^{-1} = \left(M^{-1}\right)^t$ and both are written M^{-t}. If we take $X = M^{-t}Z$, then $C_X = M^{-t}\left(M^{-t}\right)^t = M^{-t}M^{-1} = H^{-1}$, as desired. The equation $X = M^{-t}Z$ is equivalent to $M^tX = Z$. Since M is lower triangular, M^t is upper triangular. The process of finding X from Z and M to satisfy $M^tX = Z$ is called *back substitution*. Any software system that computes Cholesky decompositions also has a procedure to do back substitution. If H is $n \times n$ and not sparse, it takes $O(n^3)$ work to compute the Cholesky factor M and $O(n^2)$ work to do a back substitution.

6 Rejection sampling

6.1 Basic rejection

Rejection sampling converts samples of a *proposal* density, $g(x)$, into samples of a *target* density, $f(x)$. If you can sample g, rejection allows you to sample f. In practice, f is given and you try to find g that you can sample and that is close enough to f for rejection sampling to be efficient. Rejection sampling uses an *acceptance probabiltiy* function $A(x)$, which is a probability: $A(x) \in [0,1]$ for each x. A step of rejection sampling uses a sample $X \sim g$. This is the *proposal*. The proposal is *accepted* with probability $A(X)$. If X is accepted, it is the sample of f. If X is rejected (not accepted), you generate a new independent $X \sim g$. Rejection sampling continues this propose and accept/reject until the first acceptance. All proposals and acceptance/rejection decisions are independent of each other. Assuming `gSamp()` produces independent samples from g, the code for basic rejection sampling could be like this:

```
while(1):
    X = gs.gSamp()      # an independent sample from g
    A = Z*f(X)/g(X)     # acceptance probability, see below
    if ( uSamp() < A):
        break           # break means accept
```

We compute the probability density of an accepted sample. This is defined by $f(x)\,dx = P(X \in [x, x+dx])$, where X is a typical accepted sample. Bayes' rule gives the distribution of X conditional on acceptance. We let Z be the probability of acceptance in a given try, which is given by

$$\begin{aligned} Z &= P(\text{accept}) \\ &= \int P(\text{accept } X \mid \text{propose } X \in [x, x+dx])\,dx \\ Z &= \int A(x)g(x)\,dx\ . \end{aligned} \tag{7}$$

With this,

$$\begin{aligned}
f(x)\,dx &= P(\text{accepted } X \text{ in } [x, x+dx]) \\
&= P(\text{proposed } X \text{ in } [x, x+dx] \mid \text{accepted the proposal}) \\
&= \frac{P(\text{proposed } X \text{ in } [x, x+dx] \text{ and accepted this } X)}{P(\text{accepted})} \qquad \text{(Bayes' rule)} \\
&= \frac{g(x)\,dx \cdot A(x)}{Z} \; .
\end{aligned}$$

This leads to the important rejection sampling formula

$$f(x) = \frac{1}{Z} A(x) g(x) \; . \tag{8}$$

In practice, you know the target density f and the proposal density g. Then (8) gives

$$A(x) = \frac{Z f(x)}{g(x)} \; . \tag{9}$$

We want the largest possible acceptance probability, we we should take Z as large as possible. The constraint $A(x) \leq 1$ for all x leads to

$$Z = \max_x \frac{g(x)}{f(x)} \; . \tag{10}$$

You can think of the rejection sampling formula (8) as a thinning process. You get the graph of f starting from $\frac{1}{Z}g$ by reducing by a factor $A(x)$. This changes the shape of the distribution because the reduction factor, $A(x)$, is different in different places. A drawback is that this thinning formula only removes probability, it never adds probability. For example, if $g(x) = 0$ for some x, then $f(x) = 0$. Going further, the *tails* of g (the values of g for large x) must be large enough to create the tails of f. For example, if $g(x) = 2e^{-2x}$ and $f(x) = e^{-x}$ (and $f = g = 0$ when $x < 0$), then the tails of g are too small relative to the tails of f. The formula (8) gives $A(x) = \frac{Z}{2}e^{x}$. This is impossible if $A(x) \leq 1$ for all x.

It is common to denote normalization constants in probability densities by Z. It is also common to have a formula for a probability density with an unknown normalization constant. That means we have a formula for $h(x)$ and the desired probability density is $f(x) \propto h(x)$. This is written

$$f(x) = \frac{1}{Z} h(x) \; .$$

The normalization constant is found using the fact that f integrates to 1:

$$Z = \int h(x)\,dx \; . \tag{11}$$

In the present case, if we propose using $g(x)$ and accept using $A(x)$, then the resulting density is clearly $f(x) \propto A(x)g(x)$. The normalization constant is

$$Z = \int A(x)g(x)\,dx \; . \tag{12}$$

The optimal Z satisfies both (10) and (12). It may be that we cannot solve the optimization problem (10), but we can find a Z so that $A(x) \leq 1$ in (9) for all x. Then (12) would be satisfied, but not (10).

The *efficiency* of a rejection sampler depends on the expected number of proposals to get an acceptance. Let N be the number of proposals to get the first success. This is a geometric random variable because proposals are independent. The first trial may be an acceptance or rejection. If it is a rejection, the number of subsequent proposals needed is the same as it was before. The probability of rejection is $1 - Z$. Therefore

$$E[N] = 1 \cdot P(\text{accept}) + E[1 + N] \cdot P(\text{reject}) = Z + (1 + E[N])(1 - Z) \; .$$

Solving for $E[Z]$ gives

$$E[N] = \frac{1}{Z} \; .$$

The smaller the acceptance probability, the more proposals you need, on average, to get an acceptance.

This formula tells you how to design an efficient rejection sampler. You have a bad sampler, one with small Z, if there is an x that is much more likely in the target distribution than the proposal distribution. It is unfortunate that (12) calls for a worst case analysis rather than an average case analysis. It might be that $\frac{g}{f}$ is reasonable for most x values and yet Z is very small. Exercise 7 is an example where $\frac{g(x)}{f(x)} \to 0$ as $x \to \infty$. This leads to acceptance probability $Z = 0$.

You find a good proposal distribution by looking for a g that can be sampled and that looks like f. But even if g looks like f in the central parts of the distributions, it can fail in the tails. For example, it is possible to sample a standard normal by rejection from the *double exponential* density $g(x) = \frac{1}{2}e^{-|x|}$ (exercise 7 asks you to calculate the optimal Z), but it is not possible to sample the double exponential from a standard normal proposal because the tails of the standard normal are too thin. Of course, there are simpler direct samplers for both distributions that do not require rejection.

Suppose the target density is $f(x) \propto \sin(\pi x)$ in the interval $[0, 1]$. The normalization constant (11) is

$$\int_0^1 \sin(\pi x)\,dx = \frac{-1}{\pi} \cos(\pi x)\Big|_0^1 = \frac{2}{\pi} \; .$$

The target density is $f(x) = \frac{\pi}{2}\sin(\pi x)$. A proposal density whose graph is similar, and that we know how to sample (Subsection 4.5), is $g(x) = 6x(1 - x)$.

The efficiency is determined by

$$Z = \min \frac{6x(1-x)}{\frac{\pi}{2}\sin(\pi x)} \; .$$

The minimum presumably is achieved in the middle of the interval, $x = \frac{1}{2}$, which gives the value

$$Z = \frac{6 \cdot \frac{1}{4}}{\frac{\pi}{2}} = \frac{3}{\pi} = .955 \; .$$

This rejection sampler is a little better than 95% efficient.

Most people working on Monte Carlo have designed a rejection sampler at least once. This process can be messy, inelegant, and time consuming. But good rejection sampling is crucial for the performance of the overall code. In a lifetime practicing Monte Carlo and inventing rejection samplers, it is not likely that you will be able to make one with a 95% acceptance probability for a problem that you care about.

6.2 A multivariate example

Many multivariate distributions are approximately normal. Some of them are Gibbs Boltzmann probability densities of the form

$$f(x) = \frac{1}{Z} e^{-\beta\phi(x)} \; . \tag{13}$$

Here $\phi(x)$ is the energy in a system with configuration x, and β is the *inverse temperature.* (In equilibrium statistical physics, $\beta = 1/k_BT$, where T is the temperature in degrees above absolute zero and k_B is *Boltzmann's constant*, which is a conversion factor between temperature and energy. High temperature, which is large T, corresponds to small β.)

The *energy minimizing* configuration is the x that minimizes ϕ in (13). Denote this *equilibrium* position by x_0, and suppose it is unique and non-degenerate. A *non-degenerate* equilibrium has Hessian matrix $H = \phi''(x_0)$ that is positive definite. As the temperature goes to zero, which is the limit $\beta \to \infty$, the distribution f becomes concentrated closer and closer to x_0. For x close to x_0, we should be able to use the Taylor series approximation

$$\phi(x) \approx \phi(x_0) + \frac{1}{2}\left(x - x_0\right)^2 H \left(x - x_0\right) \; . \tag{14}$$

This approximation gives rise to the *semiclassical* approximation

$$f(x) \approx g(x) = \frac{1}{Z} e^{-\beta(x-x_0)^2 H(x-x_0)/2} \tag{15}$$

The difference between $X \sim f$ and the energy minimizing state x_0, particularly at low temperature (large β) is called *thermal fluctuation.* The semiclassical approximation is to say that at low temperatures, thermal fluctuations are approximately Gaussian with a precision matrix given by the Hessian of the energy function.

Now, suppose β is large enough that g is a good approximation to f. Could we use the semiclassical approximate distribution (15) as a proposal density for the exact (13)? There are several potential difficulties. One is that the best possible acceptance probability (10) is equal to zero. That could be if the semiclassical approximation is invalid far from x_0. Even if there is a $Z > 0$, it might be very hard to find. You might think of applying numerical optimization to (10), but that could be very expensive, particularly if we want only one sample. The curse of dimensionality may work against us too. The quadratic approximation to the Gibbs distribution may be a poor trial distribution if the dimension of x is large. This is because the next order Taylor series corrections to (14) are cubic polynomials in x. If there are n components of x, there are approximately $n^3/6$ distinct cubic monomials and third partial derivatives of ϕ. Even if each individual term is small, they may add up to something not very small.

7 Bayesian posterior

Here is a quick summary of part of Bayesian statistics. We assume there are parameters $x = (x_1, \ldots, x_d)$. We have M observations of a random variable, Y, whose probability distribution depends on the parameters:

$$Y_k \sim L(y|x) \ .$$

The PDF of Y is often called the *likelihood* function, because we are interested in how it depends on x, not y. The Bayes model is that first X is selected from a *prior* density: $X \sim \pi(x)$, then the observations are chosen to be independent samples of $L(y|X)$. The joint distribution of X and $(Y_1, \ldots, Y_M)$ is

$$\pi(x) \prod_1^M L(y_k|x) \ .$$

If the Y_k are known, the resulting conditional density of X is

$$f(x|Y_1, \ldots, Y_M) = \frac{1}{Z} \pi(x) \prod_1^M L(Y_k|x) \ . \tag{16}$$

Bayesian statistics is, essentially, the belief that the posterior distribution (16) expresses our state of knowledge about the parameters given the data. Therefore, the job of the Bayesian statistician is to produce samples from (16). This was long considered impractical because statisticians were unaware of MCMC. In the 1980's, statisticians became aware of MCMC (and invented the term). They also renamed several Monte Carlo sampling methods that we will discuss later.

Classical statistics of parameter estimation, rightly, concentrates on the degree of accuracy of statistical estimates, given the data and the probability model of how the data were generated. For example, there are *error bars*, which are intervals (called *confidence intervals* in statistics) in which the true parameter x_j is likely to lie, given the data and the probability model. Statisticians make statements like: "The probability is 90% that if the data Y_k are independent samples of $L(y|x)$, then $\widehat{X}_j - R_j \leq x_j \leq \widehat{X}_j + R_j$." The numbers $\widehat{X}_j$ and R_j are functions of the data. A Bayesian might estimate the confidence interval by creating a large number of samples of $X \sim f(x|Y_1, \ldots, Y_M)$ and taking the 5% and 95% quantiles of the values of X_j from this sample set.

But the posterior sample contains more information than just ranges for individual parameters. For example, it often happens that the data imply a strong relationship between two parameters without determining either parameter very accurately. For example, we may learn X_3+X_4 very accurately without knowing either X_3 or X_4 very accurately. A more subtle relationship between parameters might be that either $X_1 \approx X_2$ or $X_1 \approx X_3$. That is, X_1 is nearly the same as another parameter, but we don't know which. We would learn this from a posterior sample, as some of the samples would have X_1 near X_2 and other would have X_1 near X_3.

There is a family of central limit theorems for the Bayesian posterior. These are called *Bernstein von Mises* theorems. In the simple setting described above (independent samples from the same likelihood function), the theorem says that asymptotically, as $M \to \infty$, the posterior becomes approximately Gaussian. There are hypotheses, including the requirement that $\pi(x)$ (the prior) should be continuous and not rule out the correct answer. The mean is (asymptotically, not exactly) the *maximum likelihood* point, which is

$$\widehat{X}_{ML} = \arg\max_x \prod_1^M L(Y_k, x) \; .$$

This is the same (asymptotically, if π cooperates) as the *MAP* point (*maximum a-posteriori*), which is

$$\widehat{X}_{MAP} = \arg\max_x f(x|Y_1, \ldots, Y_M) \; .$$

The covariance matrix for the Gaussian is the same as the Fisher information matrix from classical maximum likelihood estimation. Therefore, if you have enough data, you can estimate the Fisher information matrix using the empirical covariance matrix of a collection of posterior samples. Also, if you have enough data, you can use the Gaussian approximation as a proposal density for the true posterior. However, my experience is that you need quite a lot of data for this to be effective. In general, it is very hard to find direct samplers of any kind for non-Gaussian multivariate distributions.

8 Weighted sampling, importance sampling

A *weighted sample* of a density f is a pair of random variables (X, W) so that X, *weighted* by W, has the density f. An informal way to say this is

$$E[W\delta(X-x)] = f(x) \tag{17}$$

for every x. More formally, if $V(x)$ is a bounded continuous function, then

$$B = \int V(x)f(x)\,dx = E[WV(X)]\ . \tag{18}$$

A weighted sampler does not have to produce $X \sim f$. The weight W compensates for the discrepancy in the X distribution. We write $(X, W) \sim f$ if (17) or (18) are satisfied.

One form of weighted sampling allows you to use a rejection sampler without rejection. You can take $X \sim g$ in (8) and

$$W = w(X)\frac{1}{Z}A(X)\ .$$

You can check the property (17) by

$$E[W\delta(X-x)] = E_g[\frac{1}{Z}A(X)\delta(X-x)] = \frac{1}{Z}A(x)E_g[\delta(X-x)] = \frac{1}{Z}A(x)g(x) = f(x)\ .$$

This is because

$$E_g[\frac{1}{Z}A(X)\delta(X-x)] = \frac{1}{Z}\int A(x')\delta(x'-x)g(x')\,dx' = \frac{1}{Z}A(x)g(x)\ .$$

Equivalently, you can check the property (18) using (8):

$$\begin{aligned} E[WV(X)] &= \frac{1}{Z}E_g[A(X)V(X)] \\ &= \frac{1}{Z}\int A(x)V(x)g(x)\,dx \\ &= \int V(x)f(x)\,dx \\ &= E_f[V(X)]\ . \end{aligned}$$

If we are sampling f to estimate the expectation (18) the procedure would be to generate N samples of g and use the estimator

$$\widehat{B} = \frac{1}{N}\sum_{k=1}^{N} w(X_k)V(X_k)\ . \tag{19}$$

An alternative would be to do rejection sampling, getting some number of exact samples of f from N proposals from g. It is an exercise to show that the weighted sampling estimate (19) has lower variance.

Unfortunately, we often are trying to sample f not to evaluate B, but for some other purpose. In that case, weighted samples may be less useful than exact samples found from the same proposal distribution using rejection.

It is common that we we can sample g and we believe g is close to f, but we cannot find the necessary Z for exact weighted sampling. That is, we have $g(x)$ and we know that $f(x) \propto w(x)g(x)$, but we do not know the normalization constant. This is the situation in Section 6.2, where we can evaluate both $g(x)$ and $e^{-\beta\phi(x)}$ easily, but we do not know Z in (13). The algorithm is to generate N samples of g, evaluate the weights W_k, then use

$$\widehat{B} = \frac{\sum W_k V(X_k)}{\sum W_k} \; .$$

9 Monte Carlo estimation and error bars

Error estimation and correctness checking are essential parts of all scientific computing. This is particularly true in Monte Carlo, where the "exact" answer always comes with some noise. Small errors in samplers can be hard to spot unless you do high precision error checking. High precision in Monte Carlo usually entails significant computing time.

Error estimation in Monte Carlo may be thought of as a problem in statistics, and many statistical ideas apply. This is true both for producing error bars in production Monte Carlo runs and for checking correctness of components of Monte Carlo codes.

9.1 Error bars, the central limit theorem

Suppose you are trying to estimate a number A, which is not random. The Monte Carlo estimate is $\widehat{A}$, which is random. An *error bar* is an estimate of the size of the difference between $\widehat{A}$ and A. One more precise version of this idea is related to what statisticians call a *confidence interval.* The interval $[\widehat{A}-\varepsilon, \widehat{A}+\varepsilon]$ is a confidence interval with *confidence* level α if

$$P(A \in [\widehat{A}-\varepsilon, \widehat{A}+\varepsilon]) \geq \alpha \; . \tag{20}$$

In this definition A is not random. The random quantities are $\widehat{A}$ and ϵ. Suppose, for instance, that our code has 95% confidence. Then there is at least a 95% chance that the code will produce numbers $\widehat{A}$ and ε that have the property that $\widehat{A}-\varepsilon \leq A$ and $\widehat{A}+\varepsilon \geq A$. The interval $[\widehat{A}-\varepsilon, \widehat{A}+\varepsilon]$ is the *error bar.* It is often represented as a bar in plots with some symbol in the center of the bar representing $\widehat{A}$. In writing, you can report the error bar as $A = \widehat{A} \pm \varepsilon$. For example, a 95% confidence interval $[4.1, 4.5]$ might be written $A = 4.3 \pm .2$.

The central limit theorem, or *CLT*, gives simple reasonably accurate error bars for most computations involving direct samplers. Suppose you want $A =$

$E_f[V(X)]$ and the estimator is

$$\widehat{A} = \frac{1}{L}\sum_{k=1}^{L} V(X_k) \; ,$$

where the X_k are i.i.d. samples of f. The CLT applies because the numbers $V(X_k)$ are i.i.d. random variables with expected value A. The number of samples, L, is likely to be large enough for the CLT to be valid if we are trying to make an accurate estimate of A. Therefore, $\widehat{A}$ is approximately normal with mean A and variance σ^2/L, where σ^2 is the variance of $V(X)$ with $X \sim f$. The *one standard deviation error bar* is a confidence interval with ε equal to the standard deviation of $\widehat{A}$, which is $\varepsilon = \sigma/\sqrt{L}$. According to the CLT, the confidence of this error bar is $\alpha = 68\%$. The custom in scientific Monte Carlo is to report such one standard deviation error bars. Others may prefer to give two standard deviation error bars $\varepsilon = 2\sigma/\sqrt{L}$. This gives 95% confidence error bars.

Usually σ^2 is unknown and must be estimated from Monte Carlo data. The standard estimator of σ^2 is

$$\widehat{\sigma^2} = \frac{1}{L}\sum_{k=1}^{L}\Big(V(X_k) - \widehat{A}\Big)^2 \; . \tag{21}$$

It is common to use $1/(L-1)$ rather than $1/L$ here. But if that makes a difference you probably don't have enough data to estimate A accurately, or to use error bars based on the CLT. If you use (21) instead of σ^2 in the error bar formulas, the α values will only be approximate. But even if you used the exact σ^2, the CLT is only a large L approximation. A wise and practical Monte Carlo expert says: "Don't put error bars on error bars." The purpose of an error bar is to know about how accurate $\widehat{A}$ is. Suppose $\widehat{A} = 2958$ and the 68% error bar is $\varepsilon = 2.543$. There doesn't seem to be much harm in reporting $A = 2958 \pm 2.3$, even though the error bar is off by 10%.

It is absolutely unprofessional to do a Monte Carlo computation without quantitative reasonably accurate error bars. You don't have to report error bars to non-technical people who would not appreciate them. But you do have to know how big they are, and to report them to any consumer of your results who has the technical training to know what they mean. For every computational assignment in this course, reasonable error bars are part of the assignment.

9.2 Histograms, verifying a sampler

All programming is error prone, and particularly scientific computing. And within scientific computing, particularly Monte Carlo. You need to verify each Monte Carlo procedure carefully. Whenever you write a sampler, you need to verify it before you put it into a larger Monte Carlo code. As with error bars, verifications are a part of every computing assignment in this class. Not just verifications of final results, but separate verifications of the component subroutines.

The histogram is a practical way to verify most direct samplers of one component random variables. A histogram divides the real axis into *bins*, $B_j = [x_j - \frac{\Delta x}{2}, x_j - \frac{\Delta x}{2})$. Here Δx is the bin size, and $x_j = j\Delta x$ is the bin center. The bin as written contains its left endpoint but not its right endpoint. Mathematically, this is irrelevant as long as the probability density is continuous. But computational floating point numbers are discrete and may sometimes land on endpoints. If the samples are $X_1, \ldots, X_L$, the *bin counts* are $N_j = \# \{X_k \in B_j\}$. N_j is the number of samples in bin B_j. A *histogram* is a graph of the bin counts. It is traditional to plot bin counts using a bar graph, but there is no scientific reason do do that.

If the X_k are samples of a probability density f, then the expected bin counts are

$$n_j = E[N_j] = Lp_j = L \int_{B_j} f(x)\, dx \; .$$

Here, p_j is the probability that a particular sample lands in B_j. In practice, it often suffices to approximate the integral by $p_j \approx \Delta x f(x_j)$, but there is no scientific reason to do this. The cost of doing the integrals more accurately is trivial compared to the cost of generating the samples.

Since $n_j \neq N_j$, you have to have an idea how much difference to expect. You need error bars for bin counts. Bin counts are binomial random variables because N_j is the sum of L independent Bernoulli random variables with the same p_j. The variance of N_j, therefore, is $\sigma^2_{N_j} = Lp_j(1-p_j)$. You can estimate p_j from the empirical bin count

$$p_j \approx \widehat{p}_j = \frac{N_j}{L} \; .$$

This is accurate if N_j is more than just a few, which it will be for lots of bins if Δx is not too small and L is large.

The histogram verification procedure would be:

1. Generate a large number of samples $X_1, \ldots, X_L$.
2. Calculate the bin counts N_j for a range of x_j containing most of the probability.
3. Calculate the error bars for N_j using $\varepsilon_j = \widehat{p}_j(1-\widehat{p}_j)\sqrt{L}$.
4. Calculate the expected bin counts n_j.
5. Graph $N_j \pm \varepsilon_j$ and n_j in the same figure. Roughly a third of the n_j should be outside the error bars.

It is a good idea to take Δx somewhat small and L very large so that you get a picture of f with error bars as small as possible.

10 Examples and exercises

1. Suppose you want $X \in \mathbb{R}^n$ uniformly distributed on the unit $n-1$ dimensional sphere. This is the same as asking for a unit vector $\|X\|_{l^2} = 1$ whose probability distribution is isotropic. You can do this by starting with any isotropic probability distribution and normalizing. Let $Z = (Z_1, \ldots, Z_n)^t$ where the Z_k are independent one dimensional standard normals (made, for example, by Box Muller). Then Z is an n dimensional standard normal with isotropic probability density $f(z) = Ce^{-\|z\|^2/2}$. The normalized random variable $X = \frac{1}{\|Z\|}Z$ is both normalized and isotropic, as desired.

2. Suppose you want X uniformly distributed in the unit ball. One approach would be to take X uniformly distributed in the cube that contains the ball. That would be $X_k = 2U_k - 1$, where the U_k are i.i.d. standard uniforms. You can then accept X if it is inside the unit ball, $\|X\| \leq 1$ and reject otherwise. Eventually you will get an acceptance. The excepted X is uniformly distributed in the unit ball. Each proposal is simple and cheap. The efficiency of the overall algorithm depends on its acceptance probability, Z. Show that Z is exponentially small in n. The conclusion is that generating a uniform in the ball by rejection from a uniform in the cube is an exponentially bad idea. *One approach*: there is a formula for the volume of the unit ball in n dimensions. The cube has side 2 and volume 2^n. The ratio of these volumes is the acceptance probability. You will need to use an asymptotic approximation of the Gamma function, such as $\Gamma(n) = (n-1)! \approx (n-1)^{n-1}e^{-n-1}$. *Another approach*: If X_k is uniform in $[-1,1]$ then $E\left[X^2\right] = \frac{1}{3}$. Therefore, in n dimensions, $E\left[\|X\|^2\right] = \frac{n}{3}$. Cramer's theorem from large deviation theory implies that $P\left(\|X\|^2 \leq 1\right)$ is exponentially small.

3. Another way to generate X uniform in the unit ball is to write $X = RY$, where $R = \|X\| \in [0,1]$, and Y is uniform on the sphere. Think of this as working in spherical coordinates. Exercise 1 lets you generate Y. R is a scalar whose CDF is $F(r) = Cr^n$ ($P(R \leq r)$ is proportional to the volume of the ball of radius r.). The constant is found from $1 = F(1) = C$. The CDF inversion method gives R with the desired CDF by solving $F(R) = U$. In this case, that is just $R = U^{1/n}$.

4. Let $K(x)$ be the Green's function for the Debye Hückel operator. This satisfies $\triangle K(x) - mK(x) = \delta(x)$. Since K is negative and decays exponentially, $-K$ can be normalized to be a probability density. The normalization constant may be found by integrating both sides over $\mathbb{R}^n$:

$$\int \triangle K(x)\,dx - m\int K(x)\,dx = \int \delta(x)\,dx$$
$$m\int \left(-K(x)\right)\,dx = 1\;.$$

To sample the probability density $f(x) = \frac{-1}{m}K(x)$, you choose an exponential T with rate constant $\lambda = m$, then you take $X \sim \mathcal{N}(0, mI)$.

5. Suppose $R > 0$ has probability density $f(r) = re^{-r^2/2}$. Show that the CDF is $F(r) = 1 - e^{-r^2/2}$. Show that if you solve the equation $F(R) = U$, you get a sampler for f that is equivalent to (5).

6. (*easy*) Show that the formula (12) gives $Z \leq 1$ for any pair of probability densities f and g. Note that there are x values with $\frac{g(x)}{f(x)} > 1$. The problem is to show that $\frac{g(x)}{f(x)} \leq 1$ for some x provided that f and g are probability densities.

7. Solve the optimization problem (10) when f is the standard normal and g is the double exponential. The efficiency should be around 75%. Consider generating a double exponential from a standard normal. Show that the optimization problem (10) leads to $Z = 0$.

8. Suppose we have a direct weighted sampler of a probability density g. This means that there is a procedure so that `[X,W] = gSampW()` produces an independent weighted sample of g in the sense of (17) or (18).

   (a) Does the rejection method of Subsection 6.1 turn (X, W) into a weighted sample of f?

   (b) Suppose $L(x) = \frac{f}{g}$ is the likelihood ratio. Is $(X, WL(X))$ a weighted sample of f?

9. Describe the mechanics of using the CLT to estimate error bars when you are using a weighted direct sampler of f. Describe how to create "weighted" histograms to verify that (X, W) is a weighted sample of f.