1 Weighted Importance Sampling

If is often the case that we wish to estimate $\ell = \mathbb{E}_f S(\mathbf{X})$ without knowing everything about the density f. For example, a density can be written as

$$f(\mathbf{x}) = \frac{h(\mathbf{x})}{Z},$$

where Z is the normalizing constant (that is $Z = \int f(\mathbf{x})\,\mathrm{d}\mathbf{x}$). In many realistic applications, we do not know Z, even if we do know h and have a way of sampling from f (it is often possible to sample from a density without knowing its normalizing constant). An obvious problem, then, is how we should carry out importance sampling in such a setting.

Weighted importance sampling is one way of addressing this problem. The normal importance sampling estimator is of the form

$$\widehat{\ell}_{\mathsf{IS}} = \frac{1}{N}\sum_{i=1}^{N} \frac{f(\mathbf{X}^{(i)})}{g(\mathbf{X}^{(i)})} S(\mathbf{X}^{(i)}),$$

where $\mathbf{X}^{(1)}, \ldots, \mathbf{X}^{(N)}$ is an i.i.d. sample from g. This returns a sample mean that is weighted by the likelihood ratios

$$\frac{f(\mathbf{X}^{(1)})}{g(\mathbf{X}^{(1)})}, \ldots, \frac{f(\mathbf{X}^{(N)})}{g(\mathbf{X}^{(N)})}.$$

The idea of weighted importance sampling is to use another weight, which is of the form

$$W(\mathbf{x}) = \frac{h(\mathbf{x})}{g(\mathbf{x})}.$$

Instead of using the standard importance sampling estimator, we now have to use an estimator of the form

$$\widehat{\ell}_{\mathsf{WIS}} = \frac{\frac{1}{N}\sum_{i=1}^{N} W\left(\mathbf{X}^{(i)}\right) S\left(\mathbf{X}^{(i)}\right)}{\frac{1}{N}\sum_{i=1}^{N} W\left(\mathbf{X}^{(i)}\right)},$$

with $\mathbf{X}^{(1)}, \ldots, \mathbf{X}^{(N)}$ an i.i.d. sample from g.

Now,

$$\begin{aligned}\mathbb{E}_g W(\mathbf{X}) S(\mathbf{X}) = \mathbb{E}_g \frac{h(\mathbf{X})}{g(\mathbf{X})} S(\mathbf{X}) &= \int \frac{h(\mathbf{x})}{g(\mathbf{x})} S(\mathbf{x}) g(\mathbf{x})\,\mathrm{d}\mathbf{x} \\ &= Z \int \frac{h(\mathbf{x})}{Z} S(\mathbf{x})\,\mathrm{d}\mathbf{x} = Z\,\mathbb{E}_f S(\mathbf{X}).\end{aligned}$$

Likewise,

$$\mathbb{E}_g W(\mathbf{X}) = \mathbb{E}_g \frac{h(\mathbf{X})}{g(\mathbf{X})} = \int \frac{h(\mathbf{x})}{g(\mathbf{x})} g(\mathbf{x})\,\mathrm{d}\mathbf{x} = Z \int \frac{h(\mathbf{x})}{Z}\,\mathrm{d}\mathbf{x} = Z.$$

Thus, by the strong law of large numbers, we get in the limit that

$$\lim_{N\to\infty} \frac{\frac{1}{N}\sum_{i=1}^{N} W\left(\mathbf{X}^{(i)}\right) S\left(\mathbf{X}^{(i)}\right)}{\frac{1}{N}\sum_{i=1}^{N} W\left(\mathbf{X}^{(i)}\right)} \to \frac{Z\,\mathbb{E}_f S(\mathbf{X})}{Z} = \mathbb{E}_f S(\mathbf{X}).$$

Note that this estimator is biased. However, the bias is $O(1/n)$, so it is not too bad in practice (and we don't always have an alternative).

2 Sequential Importance Sampling

So far, when we have considered a problem like $\ell = \mathbb{P}(X_{10} > 5)$, we have considered a process that can be written as a sum of independent random variables. For example

$$X_n = \sum_{i=1}^{n} Y_i,$$

where $Y_1, \ldots, Y_n$ are i.i.d. draws from f. In this case, we can write

$$\widehat{\ell}_{\mathsf{IS}} = \frac{1}{N}\sum_{i=1}^{N} \left(\prod_{j=1}^{10} \frac{f(Y_j^{(i)})}{g(Y_j^{(i)})}\right) \mathbb{I}\left(\sum_{j=1}^{10} Y_j^{(i)} > 5\right).$$

One advantage of this formulation is that if we can sample Y_1, calculate the likelihood ratio, then sample Y_2, update the likelihood ratio (by multiplying by $f(Y_2)/g(Y_2)$), and so on. In particular, if we wish to simulate a process until a stopping time, then we can simply stop when this stopping time is reached, without having to worry about how to calculate the joint density afterwards. When the $\{Y_j\}_{j=1}^{n}$ are dependent, things are a little more complicated. Continuing with our $\ell = \mathbb{P}(X_{10} > 5)$ example, in the case of dependent random variables, we would write

$$\widehat{\ell}_{\mathsf{IS}} = \frac{1}{N}\sum_{i=1}^{N} \frac{f(Y_1^{(i)}, \ldots, Y_n^{(i)})}{g(Y_1^{(i)}, \ldots, Y_n^{(i)})} \mathbb{I}\left(\sum_{j=1}^{10} Y_j^{(i)} > 5\right).$$

However, we can often write this in a more convenient form. Note that,

$$f(y_1, \ldots, y_n) = f(y_1) f(y_2 \,|\, y_1) \cdots f(y_n \,|\, y_1, \ldots, y_{n-1}),$$

or, in Bayesian notation (which makes things a bit more compact),

$$f(y_{1:n}) = f(y_1) f(y_2 \,|\, y_1) \cdots f(y_n \,|\, y_{1:n-1})$$

Likewise, we can write

$$g(y_{1:n}) = g(y_1) g(y_2 \,|\, y_1) \cdots g(y_n \,|\, y_{1:n-1}).$$

If we know these conditional densities, then we can write the likelihood ratio in the form

$$W_n(y_{1:n}) = \frac{f(y_1)f(y_2 \,|\, y_1)\cdots f(y_n \,|\, y_{1:n-1})}{g(y_1)g(y_2 \,|\, y_1)\cdots g(y_n \,|\, y_{1:n-1})}.$$

If we write,

$$W_1(y_1) = \frac{f(y_1)}{g(y_1)},$$

then

$$W_2(y_{1:2}) = \frac{f(y_2 \,|\, y1)}{g(y_2 \,|\, y1)}\frac{f(y_1)}{g(y_1)} = \frac{f(y_2 \,|\, y1)}{g(y_2 \,|\, y1)}W_1(y_1),$$

and, more generally,

$$W_n(y_{1:n}) = \frac{f(y_n \,|\, y_{1:n-1})}{g(y_n \,|\, y_{1:n-1})}W_{n-1}(y_{1:n-1}).$$

In cases where the Markov property holds,

$$W_n(y_{1:n}) = \frac{f(y_n \,|\, y_{n-1})}{g(y_n \,|\, y_{n-1})}W_{n-1}(y_{1:n-1}).$$

Using this formulation, we can update until a stopping time, then stop updating. This formulation also allows for sophisticated methods, such as those were certain low probability paths (i.e., paths with very small weights) are randomly killed.

3 Self-Avoiding Random Walks

Consider a random walk on a 2d lattice. That is, a Markov chain, $\{X_n\}_{n\geq 0}$, on $\mathbb{Z}\times\mathbb{Z}$ with $X_0 = 0$ and transition probabilities given by

$$\mathbb{P}\left(X_n = (k,l) \,|\, X_{n-1} = (i,j)\right) = \begin{cases} 1/4 & , \text{ if } |k-i| + |l-j| = 1 \\ 0 & , \text{ otherwise} \end{cases}.$$

Self-avoiding random walks are simply random walks that do not hit themselves.

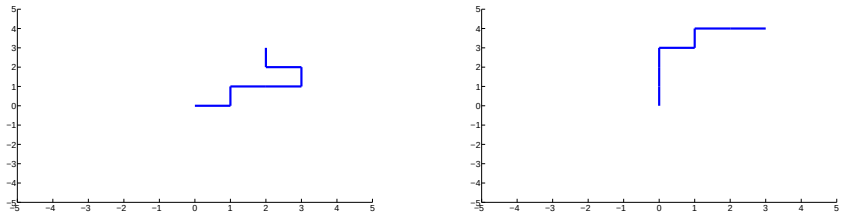

Figure 3.1: Two realizations of a self-avoiding random walk with 7 steps.

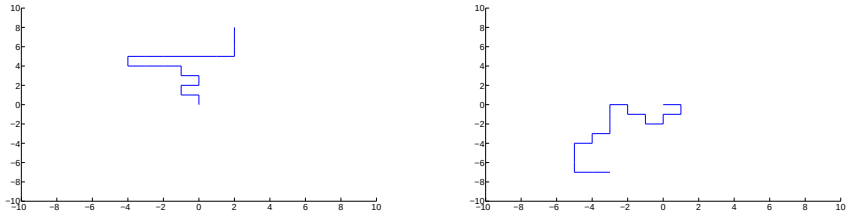

Figure 3.2: Two realizations of a self-avoiding random walk with 20 steps.

Self avoiding walks are useful as simple models of objects like polymers. They capture some fundamental behavior of strings of molecules that cannot be too close to one another, but otherwise have minimal interaction. They also appear in mathematical objects like random graphs and percolation clusters.

It is easy to generate a self-avoiding random walk of length n via Monte Carlo if n is small. We simply simulate random walks of length n until one of them is self-avoiding.

Listing 1: Matlab code

```
n = 7; i = 1;

while(i ~= n)
    X = 0; Y = 0;
    lattice = zeros(2*n + 1, 2*n+1);
    lattice(n+1, n+1) = 1;
    path = [0 0];
    for i = 1:n

        U = rand;
        if U < 1/2
            X = X + 2 * (U < 1/4) - 1;
        else
            Y = Y + 2 * (U < 3/4) - 1;
        end

        path_addition = [X Y];
        path = [path; path_addition];

        lattice_x = n + 1 + X;
        lattice_y = n + 1 + Y;

        if lattice(lattice_x, lattice_y) == 1
            i = 1; break;
        else
            lattice(lattice_x, lattice_y) = 1;
        end
    end
```

```
end

clf; hold on;
axis([-n n -n n]);
for j = 1:n
    line([path(j,1), path(j+1,1)], [path(j,2), path(j+1,2)]);
end
```

The problem is that for large n, it is very unlikely that a random walk will be a self-avoiding random walk. To put this in perspective, there are 4^n possible random walks of length n on the 2D square lattice. In general, the number of self-avoiding random walks for a given n is not known. However, for small n, these have been calculated.

- For $n = 5$, there are 284 self-avoiding walks. So, the probability that a single random walk of length 5 will be self-avoiding is

$$\frac{284}{4^5} = \frac{284}{1024} \approx 0.2773.$$

  This means we only need to generate roughly 4 walks in order to get a self-avoiding one.

- For $n = 10$, there are 441000 self-avoiding random walks. So, the probability is

$$\frac{44100}{4^{10}} = \frac{44100}{1048576} \approx 0.0421.$$

  This means we need to generate about 24 walks in order to get a self-avoiding one.

- For $n = 20$, there are 897697164 self-avoiding random walks. The probability is

$$\frac{897697164}{4^{20}} \approx 8.16 \times 10^{-4}.$$

  so we need to generate about 1125 walks in order to get a self-avoiding one.

Pretty clearly, the situation becomes unworkable by $n = 100$ or $n = 150$. Unfortunately, people are often interested in asymptotic results when considering objects like self-avoiding random walks. In order to get information about asymptotic behavior, we need to be able to generate statistics for random walks with large n values. An obvious modification to the standard algorithm would be to try to choose the next step of the random walk to avoid the places the random walk has already been. The simplest way to do this is to chose the next site of the random walk from the set of empty neighbors of the current site.

PICTURE HERE

This approach is straightforward to implement in Matlab.

Listing 2: Matlab code

```
n = 250; i = 1;
moves = [0 1; 0 -1; -1 0; 1 0];

while(i ~= n)
    X = 0; Y = 0;
    lattice = zeros(2*(n+1) + 1, 2*(n+1)+1);
    lattice(n+2, n+2) = 1;
    path = [0 0];
    for i = 1:n
        lattice_x = n + 2 + X; lattice_y = n + 2 + Y;

        up = lattice(lattice_x,lattice_y + 1);
        down = lattice(lattice_x,lattice_y - 1);
        left = lattice(lattice_x-1,lattice_y);
        right = lattice(lattice_x+1,lattice_y);
        neighbors = [1 1 1 1] - [up down left right];

        if sum(neighbors) == 0
            i = 1; break;
        end

        direction = ...
            min(find(rand<(cumsum(neighbors)/sum(neighbors))));
        X = X + moves(direction,1);
        Y = Y + moves(direction,2);

        lattice_x = n + 2 + X; lattice_y = n + 2 + Y;
        lattice(lattice_x,lattice_y) = 1;
        path_addition = [X Y];
        path = [path; path_addition];
    end
end

clf; hold on;
axis([-n n -n n]);
for j = 1:n
    line([path(j,1), path(j+1,1)], [path(j,2), path(j+1,2)]);
end
```

This approach does not solve all our problems (it is still possible to a path to die out early), however it significantly increases the length of the self-avoiding walks we are able to generate in a reasonable amount of time. Unfortunately, this approach does not generate self-avoiding walks of length n uniformly. Consider the two self-avoiding random walks of length 5 shown in figures 3.3 and 3.4. The first has probability $1/4 \times 1/3 \times 1/3 \times 1/3 \times 1/3$ and the second has probability $1/4 \times 1/3 \times 1/3 \times 1/3 \times 1/2$. Basically, the algorithm is biased towards more compact configurations. You can also see this in figure 3.5 and figure 3.5, which are less spread out than most self-avoiding walks. The obvious way to try to fix

this is importance sampling.

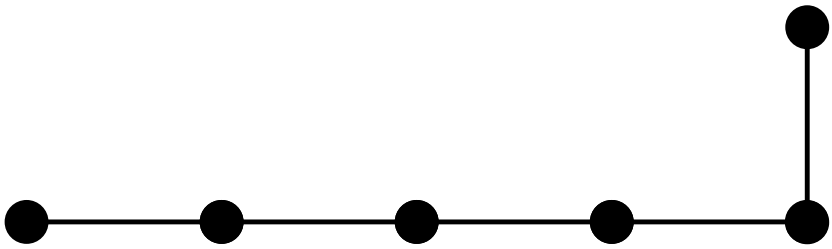

Figure 3.3: A path of a self-avoiding walk of length 5.

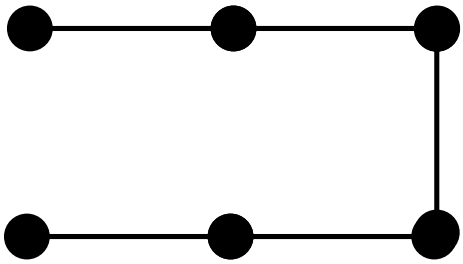

Figure 3.4: A path of a self-avoiding walk of length 5.

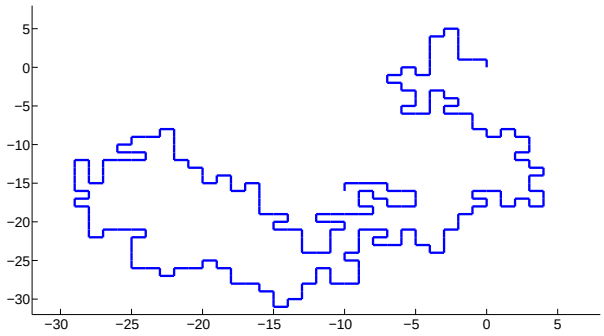

Figure 3.5: A realization of a self-avoiding random walk with 250 steps using the new technique.

The probability mass function for self-avoiding walks starting at $\mathbf{x}_0 = (x_0, y_0)$, which we represent by $\mathbf{x}_1 = (x_1, y_1), \ldots, \mathbf{x}_n = (x_n, y_n)$, is given by

$$p(\mathbf{x}_1, \ldots, \mathbf{x}_n) = \frac{\mathbb{I}\left((\mathbf{x}_1, \ldots, \mathbf{x}_n) \in E_n\right)}{Z_n},$$

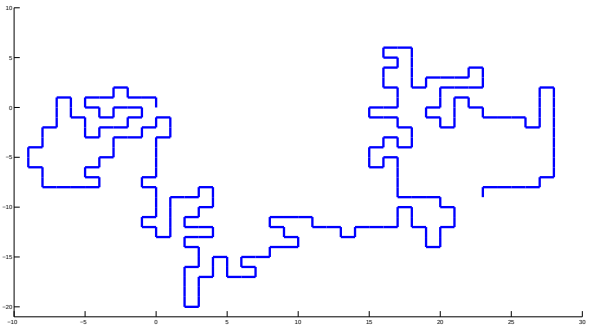

Figure 3.6: A realization of a self-avoiding random walk with 250 steps using the new technique.

where E_n is the set of self-avoiding random walks of length n. Unfortunately, as mentioned before, we do not know Z_n. However, we can use weighted importance sampling instead. To do this, we still need an expression for $q(\mathbf{x}_1, \ldots, \mathbf{x}_n)$, the probability mass function based on the new method. We can get this expression using sequential importance sampling. Note that, at step $i-1$ of the random walk, we know all the information up to step $i-1$, so we can calculate $q(\mathbf{x}_i \,|\, \mathbf{x}_0, \ldots, \mathbf{x}_{i-1})$. Let d_{i-1} be the number of unoccupied neighbors of $\mathbf{x}_{i-1}$. This is a function of $\mathbf{x}_0, \ldots, \mathbf{x}_{i-1}$. Then,

$$q(\mathbf{x}_i \,|\, \mathbf{x}_0, \ldots, \mathbf{x}_{i-1}) = \begin{cases} 1/d_{i-1}, & \text{if } \mathbf{x}_i \text{ is an unoccupied neighbor of } \mathbf{x}_{i-1} \\ 0, & \text{otherwise} \end{cases}$$

Thus, a successful realization of a self-avoiding random walk under our algorithm, $\mathbf{x}_1, \ldots, \mathbf{x}_n$ will have a probability of

$$q(\mathbf{x}_1, \ldots, \mathbf{x}_n) = \frac{\mathbb{I}\left((\mathbf{x}_1, \ldots, \mathbf{x}_n) \in E_n\right)}{d_0 \cdots d_{n-1}}$$

Note that $p(\mathbf{x}_1, \ldots, \mathbf{x}_n) \propto \mathbb{I}\left((\mathbf{x}_1, \ldots, \mathbf{x}_n) \in E_n\right)$, so we can use weights of the form

$$W(\mathbf{x}_1, \ldots, \mathbf{x}_n) = \mathbb{I}\left((\mathbf{x}_1, \ldots, \mathbf{x}_n) \in E_n\right) d_0 \cdots d_{n-1},$$

in weighted importance sampling.

3.0.1 Estimating Mean Square Extension

One of the classical objects of interest for self-avoiding random walks is mean square extension. Given a self-avoiding random walk of length n, the mean square extension is defined as $\mathbb{E}\left\|\mathbf{X}_n - \mathbf{x}_0\right\|^2$. Starting at $\mathbf{x}_0 = (0, 0)$, this is $\mathbb{E}\left\|\mathbf{X}_n\right\|^2 = \mathbb{E}\left\|X_n^2 + Y_n^2\right\|^2$. The standard Monte Carlo estimator of this would

be

$$\widehat{\ell} = \frac{1}{N} \sum_{i=1}^{N} \left\| \mathbf{X}_n^{(i)} \right\|^2,$$

where $\mathbf{X}_n^{(1)}, \ldots, \mathbf{X}_n^{(N)}$ are i.i.d. draws from $p(\mathbf{x}_1, \ldots, \mathbf{x}_n)$. The weighted importance sampling estimator, using the alternative approach, is

$$\widehat{\ell}_{\mathsf{IS}} = \frac{\frac{1}{N} \sum_{i=1}^{N} d_0^{(i)} \cdots d_{n-1}^{(i)} \left\| \mathbf{X}_n^{(i)} \right\|^2}{\frac{1}{N} \sum_{i=1}^{N} d_0^{(i)} \cdots d_{n-1}^{(i)}},$$

where $\mathbf{X}_n^{(1)}, \ldots, \mathbf{X}_n^{(N)}$ are i.i.d. draws from $q(\mathbf{x}_1, \ldots, \mathbf{x}_n)$, and the values $d_1^{(i)}, \ldots, d_n^{(i)}$ etc. are functions of the appropriate self-avoiding random walk.

An implementation of the standard Monte Carlo approach is

Listing 3: Matlab code

```
N = 10^5; n = 5;
square_extension = zeros(N,1);

for step_i = 1:N
    i = 1;
    while(i ~= n)
        X = 0; Y = 0;
        lattice = zeros(2*n + 1, 2*n+1);
        lattice(n+1, n+1) = 1;
        for i = 1:n
            U = rand;
            if U < 1/2
                X = X + 2 * (U < 1/4) - 1;
            else
                Y = Y + 2 * (U < 3/4) - 1;
            end

            lattice_x = n + 1 + X;
            lattice_y = n + 1 + Y;

            if lattice(lattice_x, lattice_y) == 1
                i = 1;
                break;
            else
                lattice(lattice_x, lattice_y) = 1;
            end
        end
    end

    square_extension(step_i) = X^2 + Y^2;
end

mean_square_extension = mean(square_extension)
```

An implementation of the importance sampling version is

Listing 4: Matlab code

```
N = 10^5; n = 150; square_extension = zeros(N,1);
moves = [0 1; 0 -1; -1 0; 1 0];

for step_i = 1:N
    i = 1;
    while(i ~= n)
        X = 0; Y = 0; weight = 1;
        lattice = zeros(2*(n+1) + 1, 2*(n+1)+1);
        lattice(n+2, n+2) = 1;
        for i = 1:n
            lattice_x = n + 2 + X; lattice_y = n + 2 + Y;
            up = lattice(lattice_x,lattice_y + 1);
            down = lattice(lattice_x,lattice_y - 1);
            left = lattice(lattice_x-1,lattice_y);
            right = lattice(lattice_x+1,lattice_y);
            neighbors = [1 1 1 1] - [up down left right];

            if sum(neighbors) == 0
                i = 1; break;
            end
            weight = weight * sum(neighbors);
            direction = ...
                min(find(rand<(cumsum(neighbors)/sum(neighbors))));
            X = X + moves(direction,1);
            Y = Y + moves(direction,2);
            lattice_x = n + 2 + X; lattice_y = n + 2 + Y;
            lattice(lattice_x,lattice_y) = 1;
        end
    end
    weights(step_i) = weight;
    square_extension(step_i) = X^2 + Y^2;
end
mean_square_extension = ...
    mean(weights'.*square_extension) / mean(weights)
```