Monte Carlo Integration for Rendering

Steve Marschner
Cornell University
CS 6630 Fall 2022

There are lots of integration problems in rendering, most of which are high-dimensional and have integrands that can't easily be written down in analytic form, let alone integrated in closed form. The most popular approach to evaluating these is to write the definite integral you want to compute as the expected value of something, then use random computational experiments to estimate that expected value. This approach is known as Monte Carlo integration.

Monte Carlo integration proceeds from the definition of expected value of a random variable. Suppose we have a random variable X on a domain D, which is distributed according to the probability density function (pdf) $p(x)$; for this I write

$$X \sim p(x).$$

Then if I evaluate some function $g(x)$ on values of the random variable, the expected value is

$$E\{g(X)\} = \int_D g(x) p(x) dx$$

The high level idea of MC integration is simple. I have some definite integral to compute:

$$I = \int_D f(x) dx$$

so I define an *estimator* $g(x)$ as follows:

$$g(x) = \frac{f(x)}{p(x)}$$

and lo and behold, the expected value of g is

$$E\{g(X)\} = \int_D f(x) dx = I$$

so evaluating g for a random x that is sampled from X (that is, is distributed according to $p(x)$) will give me a result that, on average, has the value I. Of course it has random variation to it, but if the random variation is too much I can just average the results of several independent trials of this Monte Carlo experiment to get a better estimate of I.

This leads to a really simple master algorithm that describes the basic form of many renderers:

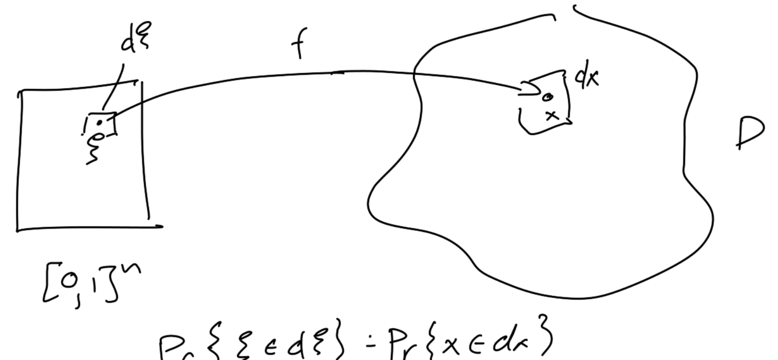
```
result = 0
for i = 1 to N
    select $x \sim p(x)$
    result += $\frac{1}{N}\frac{f(x)}{p(x)}$
```

The core operations here are (1) choosing x; (2) evaluating $f(x)$; and (3) evaluting $p(x)$. The main design choices in setting up the algorithm are how to formulate the problem as an integral and what distribution $p(x)$ to use.

1. Choosing random values

The basic tool for choosing random values is a random number generator (RNG) that generates independent, identically distributed numbers in the range $[0,1)$. I will typically call these numbers ξ, and you can assume in these notes that any variable called ξ is uniformly distributed on the unit interval. These standard uniform random numbers have a probability density of 1. It's pretty obvious how to scale such numbers to generate random values in some other interval, or how to use multiple random numbers to get points in an axis aligned rectangle, but we will want to generate random points in all kinds of weird domains that follow all kinds of weird distributions.

To do this we will invariably write some code that calls the RNG one or more times and then returns something it computed using those numbers. Mathematically, we compute a function $x = f(\xi)$ from a unit cube $[0,1)^n$ to some domain D. What is the probability density of the points generated by this process?



$$\Pr\{\xi \in d\xi\} = \Pr\{x \in dx\}$$

$$\underset{1}{\cancel{p(\xi)}}\,|d\xi| = p(x)\,dx$$

The probability density, or "pdf," $p(x)$ of x is the probability of finding x in a small region dx, divided by the size of dx. That is, $Pr\{x \in dx\} = p(x)|dx|$. Let's assume for the moment that f is a continuous one-to-one mapping—it does not map multiple different ξ to the same x. Let dx be the image of a small region $d\xi$ around ξ; then $x \in dx$ exactly when $\xi \in d\xi$, so the probabilities of these two events are the same and $p(\xi)\,|d\xi| = p(x)\,|dx|$. Since $p(\xi) = 1$, we can conclude that

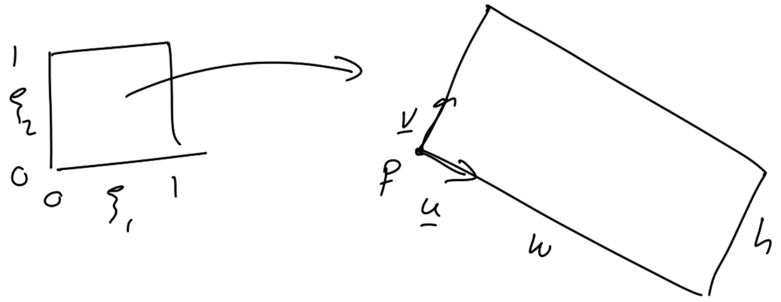
$$p(x) = \frac{|d\xi|}{|dx|} = \left|\frac{dx}{d\xi}\right|^{-1} = \left|\nabla f\right|^{-1}$$

So the density is one over the Jacobian (aka. the determinant of the derivative matrix) of f.

OK, that was a bit abstract. But if we zoom out for a minute, it says that we can analyze any code that generates random points by warping random points in the unit n-cube in a one-to-one way by looking at its derivative. If we want to generate points x in D with density $p(x)$, we just need to find a mapping from $[0,1)^n$ to D whose Jacobian is p.

Example: an affine map

Suppose f is affine; then its derivative, and hence its Jacobian, is constant, and it produces a uniform distribution. Instantiating this for 2D, suppose we generate points on a rectangle in 3D that is defined by a corner $\mathbf{p}$, an orthonormal pair of vectors $\mathbf{u}$ and $\mathbf{v}$, and its width w and height h.



$$f(\boldsymbol{\xi}) = \mathbf{p} + \xi_1 w\mathbf{u} + \xi_2 h\mathbf{v} = \mathbf{p} + [w\mathbf{u} \quad h\mathbf{v}]\,\boldsymbol{\xi}$$

Here the determinant of that matrix is wh, the area of the rectangle, so the probability density is constant $1/(wh)$ which checks out since it integrates to 1 over the rectangle.

Example: one dimensional warps

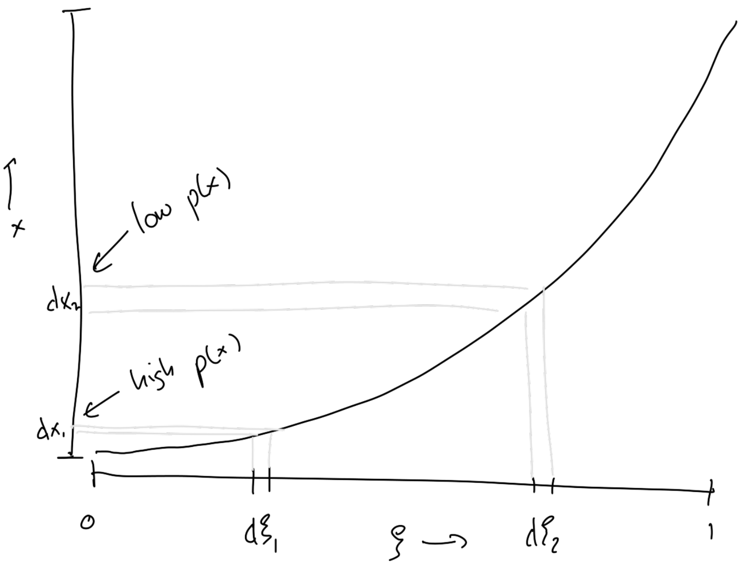
If $f : \mathbb{R} \to \mathbb{R}$ then

$$p(f(\xi)) = \frac{1}{f'(\xi)} \quad \text{or} \quad p(x) = (f^{-1})'(x)$$

This can be illustrated by looking at the images of two same-size intervals in ξ; they map to intervals in x of size proportional to $f'(\xi)$, so when f' is big, the density is small, and vice versa.

Inverse CDF sampling method

This example gives rise to a very useful method for generating samples in 1D domains. If we know $p(x)$, we can integrate to get a function whose derivative is p:



$$P(x) = \int^{x} p(x)dx$$

$$f(\xi) = P^{-1}(\xi)$$

P is the cumulative distribution function, and if we use its inverse for the warping function f, we will get samples distributed according to $p(x)$. This is a great tool for sampling 1D distributions that are sufficiently cooperative analytically that we can integrate and invert them in closed form.

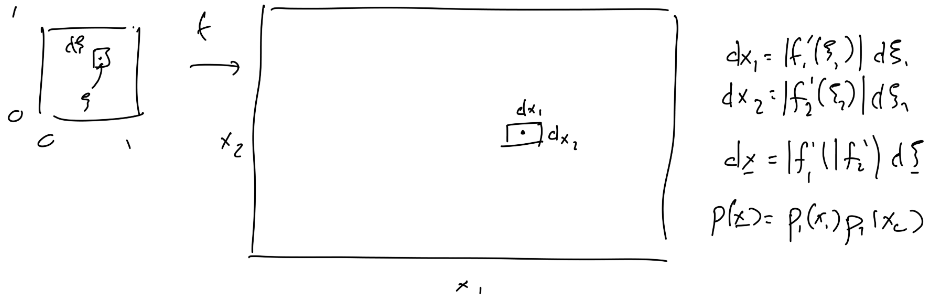
Sampling 2D domains

This method works in 1D but what about more dimensions? In graphics we very often are generating samples in 2D domains, for instance by sampling points on surfaces or directions in the hemisphere.

The easy case is when the desired 2D distribution is separable—that is

$$p(\mathbf{x}) = p(x_1, x_2) = p_1(x_1)p_2(x_2).$$

Then we can simply take a 2D random point and apply the inverse CDF method to each of the two coordinates separately. A 2D picture of this is that a little square $d\boldsymbol{\xi}$ turns into a little rectangular $d\mathbf{x} = dx_1 dx_2 = |f_1'(\xi_1)|\,|f_2'(\xi_2)|\,d\boldsymbol{\xi}$, so designing f_1 and f_2 separately achieves the desired $p(\mathbf{x})$. Another way of saying this is that x_1 and x_2 are statistically independent so they can be generated separately.



If $p(\mathbf{x})$ is not separable, we can still rescue this approach by using the marginal and conditional distributions of p. Using the notation that p with a single argument is a marginal distribution and with arguments separated by a bar is a conditional distribution:

$$p(x_1) = \int p(x_1, x_2) d x_2$$

$$p(x_2 | x_1) = p(x_1, x_2) / p(x_1)$$

First selecting x_1 with density $p(x_1)$ (the marginal density) and then selecting x_2 with density $p(x_2 | x_1)$ (the conditional density of x_2 which depends on the chosen value of x_1) will produce samples distributed according to p.

We don't see this non-separable marginal–conditional approach being used that much, possibly because the required integral is often not friendly or possibly because it tends to produce a lot of distortion in the warp (more on this later).

Radially symmetric densities

One frequent application of the separable sampling approach is to densities that are radially symmetric—that is, they depend only on distance from the center of the domain. In this case we can use a polar coordinates mapping to get a density over (r, θ) that depends only on r, making it separable (with a uniform distribution as the θ pdf). For instance, suppose we want to generate points with a uniform distribution in the unit circle. The desired pdf is the constant $p(x, y) = 1/\pi$. Because the Jacobian of the mapping from (r, θ) to (x, y) is r, the desired pdf in polar coordinates is $p(r, \theta) = r/\pi$. We can write this as a separable product:

$$p(r, \theta) = \left(\frac{1}{2\pi} \right) (2r) = p(\theta) p(r)$$

Applying the inverse CDF method to both variables separately we get

$$\begin{aligned} \theta &= 2\pi\xi_1 \\ r &= \sqrt{\xi_2} \\ (x, y) &= (r \cos\theta, r \sin\theta) \end{aligned}$$

The same process works for any radially symmetric distribution where the required integral works out neatly. This includes sampling spheres and hemispheres with uniform or cosine-proportional sampling, sampling the subtense of a spherical light source, or sampling radially symmetric microfacet distributions for BRDF sampling (these last two needed for the assignment!).

Box-Muller transformation

I want to show you one nifty and maybe surprising application of this idea, proposed by Box and Muller in a 1958 *paper*. Suppose I have the (quite common) problem of generating samples distributed according to a standard normal (Gaussian) distribution:

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

You might try the inverse CDF method but you will run aground because the CDF doesn't have a closed form. Though functions to compute it ("erf") are available in most math libraries, the inverse is less universally available and is a bit *complicated* to compute.

A (the?) nifty feature of the Gaussian distribution is that in multiple dimensions it's both separable and radially symmetric. That is, if I define a 2D distribution by making the two coordinates independently Guasssian, I get

$$p(x, y) = \frac{1}{2\pi} e^{-x^2/2} e^{-y^2/2} = \frac{1}{2\pi} e^{-(x^2+y^2)/2} = \frac{1}{2\pi} e^{-r^2/2}$$

Sampling this as a separable distribution in x and y didn't work out so well but let's try sampling it in polar coordinates. This leads to

$$\begin{aligned} p(r, \theta) &= \frac{1}{2\pi} r e^{-r^2/2} \\ p(r) &= r e^{-r^2/2} \\ P(r) &= \int_0^r p(t) dt = 1 - e^{-r^2/2} \end{aligned}$$

This time the integral worked out just great (with the additional factor of r in there), and it's simple to invert leading to the sampling procedure

$$\begin{aligned} \theta &= 2\pi \xi_1 \\ r &= \sqrt{-2 \ln(1 - \xi_2)} \\ x &= r \cos \theta \\ y &= r \sin \theta \end{aligned}$$

Computed in this way, x and y are independently and normally distributed. It's more traditional to replace $1 - \xi_2$ with just ξ_2, which has the same density (this may provide better precision for large values of x but does require a special case for $\xi_2 = 0$). So at the cost of generating two random numbers, I get two Gaussian-distributed values. I can sample from an n-dimensional Gaussian by doing this ceil(n/2) times.

This is a neat example where making the problem "harder" (sampling a 2D gaussian rather than 1D) actually made it easier.

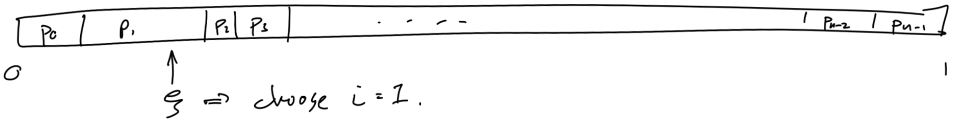
Aside: Someone pointed out after class that this provides a way to generate random points on the unit n-sphere, by first generating Gaussian-distributed (n+1)-D points and then normalizing them to unit length—pretty cool. For the 2-sphere in 3-space we have easy procedures already that use two random numbers (rather than four), but in higher dimensions (e.g. generating random unit quaternions) it might be more arduous to work out the parameterizations and this approach is simple!

Discrete distributions

Sometimes a distribution is hard to deal with and we resort to storing tables of precomputed values; or perhaps data that defines a particular probability distribution is measured or computed and arrives in the form of an array of samples (like an image, texture map, or environment map). In these cases we need to be able to sample from distributions defined by discrete collections of values.

A simple example of this is where we have a 1D distribution that is defined by a list of N probabilities that sum up to 1. We would like to select an integer in the range $[0, \mathrm{N})$ for which the probability of selecting the integer i is equal to the table entry $p[i]$ (using C-style indexing here).

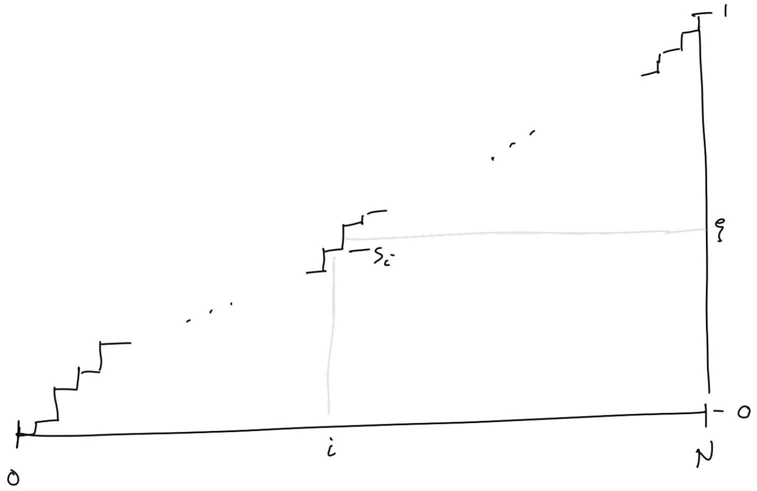
This problem can be solved by the discrete analog of the inverse CDF sampling method above. Think of the N probabilities as bars that stack up to fill the unit interval:



then take a uniformly random ξ and choose the bar that contains ξ. This amounts to finding i such that $s_i \le \xi < s_{i+1}$, where the s are the partial sums

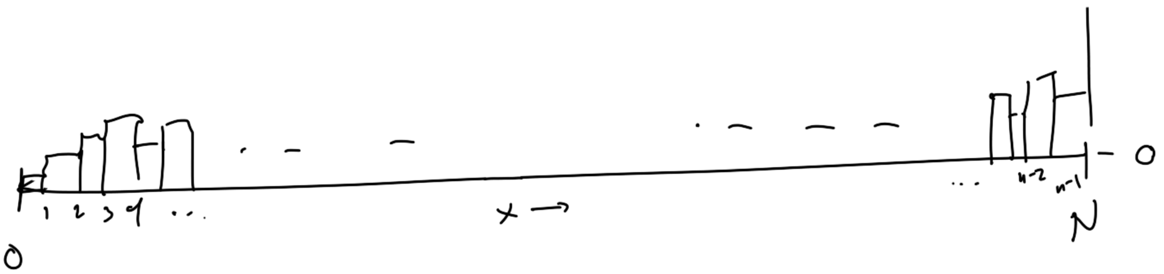
$$s_i = \sum_{j<i} p_j$$

(in particular $s_0 = 0$ and $s_N = 1$). You can think of these partial sums as playing the role of the CDF in the continuous case, leading to a picture like this:

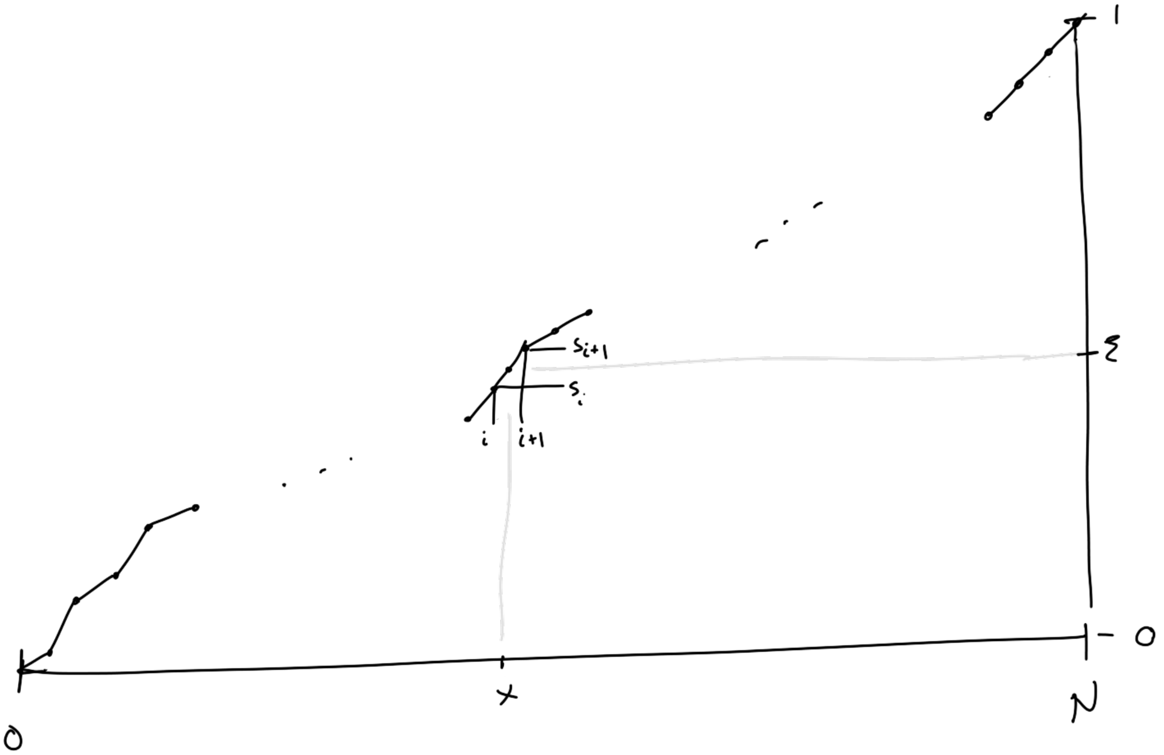


You can code this up efficiently by storing the partial sums in an array, then using binary search to find i given ξ.

This has been about choosing an integer with probability p_i, but what if I was really thinking of my probability table as expressing a piecewise constant pdf on the interval of real numbers from 0 to N? The graph of this density is like this:



The cdf that goes with this pdf is piecewise linear, and applying the inverse-CDF method to it gives a two-step algorithm that first selects a linear segment in the same way as above, then does linear interpolation to decide where x goes in that segment:



$$x = i + \frac{\xi - s_i}{s_{i+1} - s_i}$$

This is an example of a general pattern that shows up in many places. We are sampling a domain that is a union of a bunch of continuous pieces (here they are the intervals between intervals), and so we first choose which piece to sample by letting ξ fall into one of several bins. Then we use the relative position of ξ within that bin as a random number to choose a point within that bin.

This pattern also applies to many other discrete/continuous choices; for example:

- choosing a light source from several sources, then choosing a point on the source
- choosing a component (specular, diffuse, mirror) of a BRDF, then sampling that component
- choosing the length of a path, then constructing a random path with that length

2. Statistical software testing

When writing software we want unit tests. But when testing Monte Carlo software we have the problem that its outputs are not deterministic. How to decide whether the output is correct when it is not deterministic?

One good answer is to use the same statistical tests used in experimental science to decide when the data produced by some experiment supports a hypothesis or could be plausibly explained by randomness. The difference is that in software testing we are usually rooting for the absence of a statistically significant effect!

Student's t-test

This standard test is used to analyze a collection of samples from a presumed normal distribution and decide whether they come from a distribution whose mean is different from a given value or not. The "null hypothesis" is that the mean is not different (e.g. participants in my drug trial who got the treatment had the same outcomes as those who didn't), and the test produces a number which is the probability with which the difference of the sample mean from the reference would be at least as large as it was if the underlying mean was actually the same. A low probability indicates a significant effect (i.e. it's implausible to explain the results by chance).

In the software testing context, we could use the same test to check something like whether a particular pixel has the right value. We would compute the pixel value several times (being careful to use different random number seeds to ensure independence), and then use Student's test to decide whether there is a statistically significant difference from the known reference answer. A significant effect indicates that with high probability you have a bug; lack of significance says that it's plausible your code is correct (you can run with more samples to check more stringently).

Chi-squared test

Sometimes, rather than a single random value, I have samples from a distribution and I want to know if they come from the right distribution. The χ^2 test is another classic statistical test that answers a discrete version of this problem: I have a number of samples that fell into a number of buckets, and I think I know the probabilities of all the buckets (and therefore the expected count of samples in each one). Of course the counts will not exactly equal the expected counts. The χ^2 test can tell me the probability of the difference between the expected counts and the actual counts being at least as large as was observed. If that probability is low, there is a significant difference (i.e. the null hypothesis that the probabilities actually are what I expected is not a plausible explanation).

In Monte Carlo sampling code this test is useful for verifying the distribution produced by some sampling procedure: divide the domain up into finite pieces, compute the probability of each piece by integrating the pdf, generate a bunch of samples and count them up in the bins. Analyze the counts looking for a statistically significant deviation from the correct probabilities; if there is, you likely have a bug; otherwise it's plausible that your code is correct.

3. Estimating Illumination Integrals

One can make an estimator with uniform solid angle, but it's rarely used. Normally we want to importance sample *something* from the integrand.

Cosine weighting

(see slides)

BRDF sampling

(see slides)

Light source sampling

(see slides)

4. Low discrepancy sampling

The presentation of Monte Carlo integration in terms of independent random samples is simple and appealing, and it is very useful to rely on the intuitions that you gain this way. But in practice we don't really use independent random samples, particularly for the "easier" parts of the problem, because more nicely distributed samples can substantially reduce variance.

Monte Carlo integrators average many samples to get an estimate of the expected value of an estimator. If the estimator is g, we can think of the output of this process as another estimator (a macro-estimator?)

$$G_N = \frac{1}{N} \sum_{i=1}^{N} g(x_i) \quad \text{where } x_i \sim p$$

Our intuition is that the variance of G_N is less than the variance of g by itself. This is simple to show, since the samples are independent and therefore their variances add:

$$\sigma^2 \left\{ \sum_{i=1}^{N} g(x_i) \right\} = \sum_{i=1}^{N} \sigma^2\{g\} = N\sigma^2\{g\}$$

The variance of G_N is $1/N^2$ times the variance of this sum:

$$\sigma^2\{G_N\} = \frac{\sigma^2\{g\}}{N}$$

The variance is inversely proportional to the number of samples. But a more intuitive error metric, measured on the same scale as g, is the standard deviation, which is the square root of the variance. The bottom line is

$$\sigma\{G_N\} = \frac{\sigma\{g\}}{\sqrt{N}}$$

This formula is the big bummer of Monte Carlo integration. To double your precision you need to quadruple the sample budget (and therefore the CPU or GPU time). To get one more significant figure of accuracy you need 100 times as much work!

The news is bad, but not always this bad. It turns out that this convergence rate can be improved in many cases

Stratified sampling

If averaging a bunch of samples with equal variance gives us this $N^{-0.5}$ convergence, maybe we can find a way to average samples that each have lower variance. One way to do this is with *stratified sampling*—rather than estimating the whole integral N times, break the integral into N pieces called "strata" and estimate each one with one sample. To keep the analysis simple I'll work out the case where we are using a uniform sampling density $p(x)$ and the strata are all the same size (and therefore the same probability). We break the domain up into N equal-size pieces called D_i:

$$I_i = \int_{D_i} f \quad ; \quad D = \bigsqcup_{i=1}^{N} D_i \quad ; \quad I = \sum_{i=1}^{N} I_i \quad ; \quad |D_i| = \frac{|D|}{N}$$

We estimate I_i with one sample x_i, using the probability density p restricted to D_i.

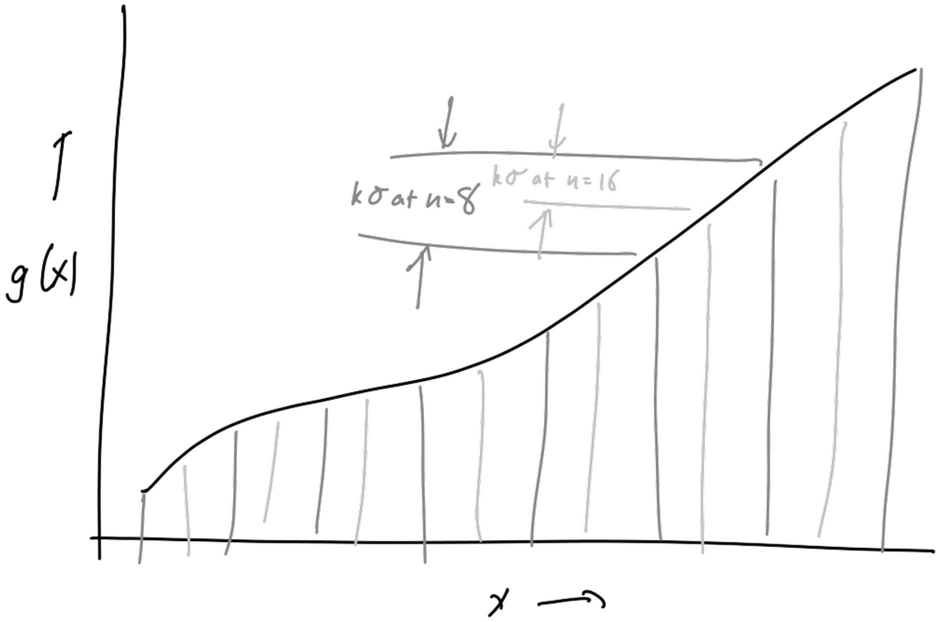
$$x_i \sim Np\,|_{D_i} \quad ; \quad g_i(x_i) = \frac{f(x_i)}{p_i(x_i)} = \frac{g(x_i)}{N} \quad ; \quad E\{g_i(x_i)\} = I_i$$

(N is the normalization factor to make p_i normalized over ($1/N$) of the domain.) Because these estimators are for the integrals over pieces of the domain, their sum (rather than their average) is the estimator for the whole integral:

$$G'_N = \sum_{i=1}^{N} g_i(x_i) = \frac{1}{N} \sum_{i=1}^{N} g(x_i) \quad ; \quad E\{G'_N\} = E\{G_N\} = I$$

Note that the formula for G'_N is the same as for G_N; this means the code to compute them is the same; we just use different samples.

There can be a difference in variance, though, if the function is smooth enough that the variance over each stratum is substantially smaller than the variance over the whole domain. If f is sufficiently smooth that it can be approximated as locally linear, then doubling the number of strata will halve the standard deviation within each stratum:



Once we have enough strata that things look locally linear, the distribution of values of g over one stratum keeps the same shape as the strata shrink, only with the range of values scaled by $1/N$. (In this 1D example the distribution of g over a small stratum is uniform.)

Now the variance of each $g(x_i)$ decreases as $1/N^2$ so the variance of the sum of N samples looks like

$$\sigma^2\left\{\sum_{i=1}^{N} g(x_i)\right\} \approx \frac{K}{N}$$

Note that this is only expected to hold once the strata are small enough that the smooth integrand can be approximated as locally linear, and the constant K depends on the properties of the integrand. (For example, if the integrand was linear, K would be $\sigma^2\{g\}$ since the linear approximation would hold right from the beginning.)

Then the variance and standard deviation of G'_N (remember dividing by N decreases the variance by N^2) are:

$$\sigma^2\{G_N\} \approx \frac{K}{N^3} \quad ; \quad \sigma\{G_N\} \approx \frac{\sqrt{K}}{N^{1.5}}$$

This is a really nice improvement in convergence rate! We can easily confirm it on a 1D example (see notebook). But before we get too excited about this let's remember it is a best-case result for very smooth integrands, and the variance reduction is less in higher dimensions. Again using the locally-linear idea, the variance scales as the square of the diameter of the strata, which is $1/\sqrt{N}$ in 2D rather than $1/N$ (and generally $1/\sqrt[d]{N}$ in d dimensions). Working this through to the final convergence rate, we get that stratified sampling can give us

$$\text{in 2D: } \sigma\{G_N\} \approx \frac{K}{N} \qquad \text{in } d \text{ dims: } \sigma\{G_N\} \approx \frac{K}{N^{1/2+1/d}}$$

in the best case of a very smooth integrand. This still makes a big difference in 2D, and the same analysis also applies for higher dimensional integrands when most of the variation of a particular integrand is aligned with a 2D subspace, which is a common case.

For these reasons most rendering systems include some mechanism to improve the distribution of samples across 2D domains of interest, such as the hemisphere, a light source, the image plane, the camera aperture, etc. Since we already generate samples by warping random points generated in a unit cube, stratification is easy to add, at least in principle: you just start with a stratified pattern of samples, and feed them to the warping functions. The warps and all the code that uses the samples can remain blissfully unaware that someone is helping them converge faster by fiddling with the samples.

Blue noise sampling

One way of thinking about the benefits of stratification is that it produces points that are spread out more nicely over the domain. Statistically independent samples are surprisingly "clumpy" and stratification improves this. There is also a signal processing viewpoint on Monte Carlo integration in which the goal of designing random sampling patterns is expressed in terms of their Fourier spectra: we want sampling patterns that have their frequency content at high spatial frequencies, and not low ones, so that when we average over the domain the random noise is at high spatial frequencies that are effectively removed by averaging. This view is particularly salient for the problem of antialiasing, or averaging over pixel areas: a good sampling pattern pushes the noise to high frequencies in the image plane which are effectively filtered out by the antialiasing filter.

This view leads to a whole subfield of "blue noise" sampling, which looks for efficient ways to generate patterns of samples that are uniformly distributed spatially (or have some specified distribution) and have little frequency content at low spatial frequencies. The name "blue" just comes from the analogy to light: blue colors contain mainly high frequency (short wavelength) radiation, so we call patterns that contain mainly high frequencies "blue." Poisson disk patterns are an example of this type of pattern, but there are very many methods for generating blue-noise patterns.

Quasi Monte Carlo

Yet another way to think about "nice" distribution of samples is discrepancy: the maximum difference between the volume of a box that is a subset of $[0,1]^d$ and the fraction of samples it contains. Low discrepancy patterns show similar benefits to stratified patterns. But patterns don't need to be random to exhibit low discrepancy. This gives rise to the field of Quasi Monte Carlo sampling, which seeks efficient algorithms to generate deterministic point sets that have low discrepancy (which generally means they will also tend to look random and blue-noise in the sense that they don't exhibit obvious spatial structure).

QMC sequences are popular in rendering partly because they solve an annoying problem that is shared by stratified and most blue noise patterns: you have to know how many samples you will need up front. Randomizing the order of the samples makes it safe to use only a prefix of the whole list of samples, in the sense that you will still get the right mean, but the variance benefits of stratification are quickly destroyed by leaving out even just a few samples. This requirement is fine in simple cases, but as things get more complex it is often awkward to ensure that the number of samples that will be needed is always known up front. In this case QMC is very appealing, because any subsequence of a QMC sequence has the same low-discrepancy properties. For this reason QMC is often the default choice for "random" sample generation in Monte Carlo renderers.

Preserving sample patterns

The benefits of stratified samples depend on having compact strata where the integrand can be expected to exhibit less variance, and similarly the properties of blue noise or low discrepancy patterns are defined in terms of "nice shaped" regions in the unit cube. When the samples are mapped into other domains with warping functions to achieve particular densities, the strata are distorted from their original shape, and if the distortion is extreme, the benefits of stratification can be eroded.

This leads to an additional design goal for warping functions: not only should they have the right Jacobian (to achieve the correct probability density) but they should have low distortion. This low distortion goal is in the background of many design choices for warping functions, for instance:

- Polar coordinates warps are convenient but do tend to produce a lot of distortion around the pole. There is a popular alternative "concentric" warp proposed by Shirley and Chiu, which is a little more code but produces a lower distortion map that does tend to make sampling patterns perform better.

- In sampling environment maps, some obvious methods fail to preserve stratification much at all. For instance, you could think of the map as a collection of pixels, use the first random number to choose a pixel, then scale that first one and use it and the second number to choose a point within the pixel. This will pretty much completely destroy the structure of your sample pattern since the strata will be shredded across scan lines of the environment map. Two better approaches are described by Matt Pharr in this post: one is to apply the marginal-conditional approach to the two dimensions of the environment map; the other is the hierarchical sample warping scheme from the optional component of our assignment. Both preserve stratification better, though neither is perfect (and indeed perfection is not a useful concept here!).

Software for samplers

The strategy of generating samples by warping from the unit cube is reflected in the software architecture of renderers.

If we only wanted to use independent random numbers all the time, all we would need is a random number generator that could be called from anywhere in the code where a random number is needed. The mappings from the unit cube to our various integration domains would exist just on paper to derive the expressions we type into the code. There are two things to think about in this strategy (which continue to be relevant below). One is seeds and determinism: generally speaking it is very helpful to be able to re-run a rendering with the same random numbers, particularly for debugging. Bugs that happen randomly are hard to catch, but once you observe one if you can then fix the random number seed in the future you will be able study the particular execution path that led to the bug. On the other hand it's very important that the random number seeds be different for different pixels so that we don't get correlated noise that becomes visible in the image. It's also important for seeds to be different if you are hoping to run your program multiple times in parallel and average the results to reduce noise; and you probably want seeds to be different across animation frames so you don't get a distracting combination of noise that is the same between frames and noise that is different depending on what has changed in the scenes.

Renderers are multithreaded programs, and to achieve determinism it's not sufficient to set a single seed, since things may not get computed in the same order from one run to the next, and in any case it's generally not safe to call a single RNG in multiple threads. For this reason you normally want a separate RNG for each thread, and you want to seed them in a way that depends deterministically on which part of the image the thread is computing. For instance you might set the seed for each pixel by using a hash of the x and y pixel indices with a global seed.

All the types of variance-reducing sample patterns that I've talked about can be abstracted as different algorithms for generating points in the unit cube. The need to preserve structured sample patterns leads to the need to expose this unit cube idea in the code: when you use a random number it matters which dimension of the cube it belongs to. Typically there is an object that is responsible for keeping track of the sample patterns being generated; Nori's Sampler class, which is a simplified version of the corresponding class in PBRT, is a typical example (see header file).

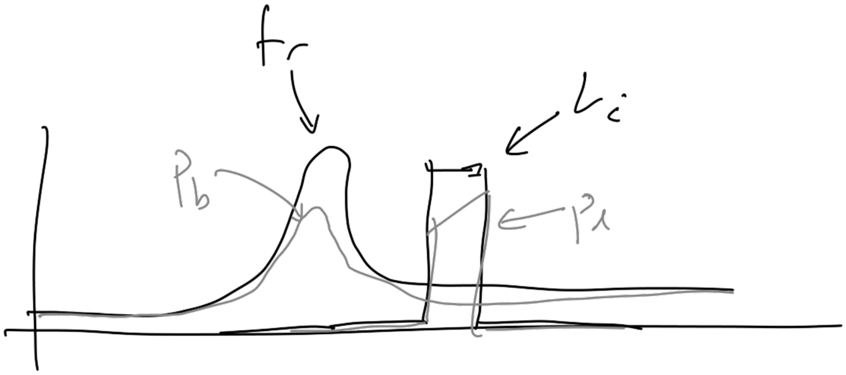
5. Multiple importance sampling

In computing local illumination (and in other cases) we have an integration problem with at least two plausible sampling strategies: sampling the BRDF and sampling the lights. In some cases one of these works pretty well, in some cases the other. For instance, rendering on an overcast day works pretty well with BRDF sampling: this will remove most of the variation from the integrand, since the incident radiance doesn't vary that much. On the other hand, if our scene has all diffuse surfaces then sampling light sources works really well (if we do it well); it leaves only the variation in the integrand due to the cosine factor, which is relatively minor.

But there are cases where neither approach works well—for instance on a sunny day in a scene where surfaces are sometimes shiny. In that case we will find that in some areas of the image the light source sampling works best whereas in others the BRDF sampling is best. In the 90s this was a big problem for renderers and people tried all kinds of heuristics to switch sampling schemes, but now there is a general solution available that works well enough in a wide range of situations, known as multiple importance sampling, or MIS. The technique and the name come from a paper by Eric Veach in 1995.

MIS is about intelligently taking weighted averages of the estimates from different estimators, in a way that helps ensure variance will be under control as long as any of the estimators is good for a particular feature of the integrand. Although there is a bit more to MIS, the simplest form of it, which Veach called the "balance heuristic" can be understood in simple terms, and that's what I'll describe here.

Suppose we have an integrand that is a product of two functions f_r and L_i which can sometimes be very peaky (I'm thinking of BRDFs and light sources but drawing in 1D):



We have pdfs p_b and p_l available that are good for sampling either of these things (maybe approximately). But in cases where most of the variation comes from one or the other of these, using the wrong pdf will produce very high variance since it hits high values with low probability. To avoid having to choose, why not use the average of these two pdfs? This is easy to implement: flip a coin, then sample from one of the two pdfs. The pdf of samples generated by this process is $(p_b + p_l)/2$ and the corresponding estimator is:

$$g_m(\omega) = \frac{f(\omega)}{p(\omega)} = 2\frac{L_i(\omega)\, f_r(\omega)\, \omega \cdot \mathbf{n}}{p_b(\omega) + p_l(\omega)}$$

This is different from the two estimators I would otherwise have to choose between:

$$g_b(\omega) = \frac{f(\omega)}{p_b(\omega)} \quad ; \quad g_l(\omega) = \frac{f(\omega)}{p_l(\omega)}$$

Intuitively, if the integrand can only be high when one of the two factors is high, and the two pdfs are proportional to the two factors, then whenever we have a high value of f, one or the other of p_b or p_l will be high, so that we don't get the high-value-over-low-pdf that leads to the outlier sample values that produce high variance in images. Averaging the two separate estimators with equal weights would not achieve this.

Weighted combinations of samples

With MIS it is common to take several samples at once, for instance one from each of two estimators, and then combine them. This is exactly the same as sampling the combined distribution twice, using a stratified random choice to decide which one to select. If we look at what happens here, the combined estimator is the average of two samples:

$$\frac{1}{2}\left(2\frac{f(\omega_1)}{p_1(\omega_1)+p_2(\omega_1)}+2\frac{f(\omega_2)}{p_1(\omega_2)+p_2(\omega_2)}\right)=\frac{f(\omega_1)}{p_1(\omega_1)+p_2(\omega_1)}+\frac{f(\omega_2)}{p_1(\omega_2)+p_2(\omega_2)}$$

We can write this as a weighted average of the two separate estimators:

$$\left(\frac{p_1(\omega_1)}{p_1(\omega_1)+p_2(\omega_1)}\right)\frac{f(\omega_1)}{p_1(\omega_1)}+\left(\frac{p_2(\omega_2)}{p_1(\omega_2)+p_2(\omega_2)}\right)\frac{f(\omega_2)}{p_2(\omega_2)}$$

or

$$w_1(\omega_1)\frac{f(\omega_1)}{p_1(\omega_1)}+w_2(\omega_2)\frac{f(\omega_2)}{p_2(\omega_2)}$$

Since we can see that $w_1 + w_2 = 1$, this is a convex linear combination of the two estimators, and since both estimators have the same mean (i.e. the value of the integral), the average also has that mean. So it is a valid estimator for our problem.

Indeed, any other scheme for selecting weights, as long as they sum to 1, will produce a valid estimator. Veach explores several of these, the most successful of which is the "power heuristic":

$$w_i^{\text{power}}=\frac{p_i^\beta}{\sum_j p_j^\beta}$$

The original one (with $\beta = 1$) is called the "balance heuristic."

Implementation in practice

The desire to implement MIS leads to a feature of sampling code that wasn't needed before MIS. In a basic single-estimator setup, you need a sample and the value of the sampling pdf at the same point, never at a different point. But for MIS the complete computation requires the value of the pdf that *did* generate the sample and also the value of the other pdf that *might have* generated that sample.

So the approach you really should use to compute direct illumination is to use MIS with sampling functions that match the surface reflectance and some approximation of the incident light.