CHAPTER THIRTEEN

13 MONTE CARLO INTEGRATION I: BASIC CONCEPTS

Before we introduce the `SurfaceIntegrators` and `VolumeIntegrators` that compute radiance along rays arriving at the camera, we will first lay some groundwork regarding the techniques they will use to compute solutions to the integral equations that describe light scattering. These integral equations generally do not have analytic solutions, so we must turn to numerical methods. Although standard numerical integration techniques like trapezoidal integration or Gaussian quadrature are very effective at solving low-dimensional smooth integrals, their rate of convergence for the higher-dimensional and discontinuous integrals that are common in rendering is poor.

Monte Carlo numerical integration methods provide one solution to this problem. They use randomness to evaluate integrals with a convergence rate that is independent of the dimensionality of the integrand. In this chapter, we review important concepts from probability and lay the foundation for using Monte Carlo techniques to evaluate the key integrals in rendering. The following chapter will then describe techniques for improving the convergence rate of these approaches and will describe the specific algorithms used in `pbrt`.

Judicious use of randomness has revolutionized the field of algorithm design. Randomized algorithms fall broadly into two classes: *Las Vegas* and *Monte Carlo*. Las Vegas algorithms are those that use randomness but always give the same result in the end (e.g., choosing a random array entry as the pivot element in Quicksort). Monte Carlo algorithms, on the other hand, give different results depending on the particular random numbers used along the way, but give the right answer *on average*. So, by averaging the results of several runs of a Monte Carlo algorithm (on the same input), it is possible to find a result that is statistically very likely to be close to the true answer. Motwani and

Raghavan (1995) have written an excellent introduction to the field of randomized algorithms.

Monte Carlo integration[1] is a method for using random sampling to estimate the values of integrals. One very useful property of Monte Carlo is that one needs only to be able to evaluate the integrand at arbitrary points in the domain in order to estimate the value of its integral $\int f(x)\,\mathrm{d}x$. This property not only makes Monte Carlo easy to implement but also makes the technique applicable to a broad variety of integrands, including those containing discontinuities.

Many of the integrals that arise in rendering are difficult or impossible to evaluate directly. For example, to compute the amount of light reflected by a surface at a point, Equation (5.8), we must integrate the product of the incident radiance and the BSDF over the unit sphere. Because object visibility and thus the incident radiance function at a point in a complex scene varies in difficult-to-predict ways, a closed-form expression for all of the terms in this product is almost never available, and even if it were, performing the integral analytically is generally not possible. Monte Carlo integration makes it possible to estimate the reflected radiance simply by choosing a set of directions over the sphere, computing the incident radiance along them, multiplying by the BSDF's value for those directions, and applying a weighting term. Arbitrary BSDFs, light source descriptions, and scene geometry are easily handled; evaluation of each of these functions at arbitrary points is all that is required.

The main disadvantage of Monte Carlo is that if n samples are used to estimate the integral, the algorithm converges to the correct result at a rate of $O(n^{-1/2})$. In other words, to cut the error in half, it is necessary to evaluate four times as many samples. In rendering, each sample generally requires that one or more rays be traced in the process of computing the value of the integrand, a computationally-expensive cost to bear when using Monte Carlo for image synthesis. In images, artifacts from Monte Carlo sampling manifest themselves as noise—pixels are randomly too bright or too dark. Most of the current research in Monte Carlo for computer graphics is about reducing this error as much as possible while minimizing the number of additional samples that must be taken.

13.1 BACKGROUND AND PROBABILITY REVIEW

We will start by defining some terms and reviewing basic ideas from probability. We assume that the reader is already familiar with basic probability concepts; readers needing a more complete introduction to this topic should consult a textbook such as Sheldon Ross's *Introduction to Probability Models* (2002).

A *random variable* X is a value chosen by some random process. We will generally use capital letters to denote random variables, with exceptions made for a few Greek symbols that represent special random variables. Random variables are always drawn from some domain, which can be either discrete (e.g., a fixed set of possibilities) or continuous (e.g.,

1 For brevity, we will refer to Monte Carlo integration simply as "Monte Carlo."

the real numbers $\mathbb{R}$). Applying a function f to a random variable X results in a new random variable $Y = f(X)$.

For example, the result of a roll of a die is a discrete random variable sampled from the set of events $X_i = \{1, 2, 3, 4, 5, 6\}$. Each event has a probability $p_i = \frac{1}{6}$, and the sum of probabilities $\sum p_i$ is necessarily one. We can take a continuous, uniformly distributed random variable $\xi \in [0, 1)$ and map it to a discrete random variable, choosing X_i if

$$\sum_{j=1}^{i-1} p_j < \xi \le \sum_{j=1}^{i} p_j .$$

For lighting applications, we might want to define the probability of sampling illumination from each light in the scene based on the power Φ_i from each source relative to the total power from all sources:

$$p_i = \frac{\Phi_i}{\sum_j \Phi_j}.$$

Notice that these p_i also sum to one.

The *cumulative distribution function* (CDF) $P(x)$ of a random variable is the probability that a value from the variable's distribution is less than or equal to some value x:

$$P(x) = Pr\{X \le x\}.$$

For the die example, $P(2) = \frac{1}{3}$, since two of the six possibilities are less than or equal to 2.

13.1.1 CONTINUOUS RANDOM VARIABLES

In rendering, discrete random variables are less common than continuous random variables, which take on values over ranges of continuous domains (e.g., the real numbers or directions on the unit sphere).

A particularly important random variable is the *canonical uniform random variable*, which we will write as ξ. This variable takes on all values in its domain $[0, 1)$ with equal probability. This particular variable is important for two reasons. First, it is easy to generate a variable with this distribution in software—most run time libraries have a pseudo-random number generator that does just that.[2] Second, as we will show later, it is possible to generate samples from arbitrary distributions by first starting with canonical uniform random variables and applying an appropriate transformation. The technique described previously for mapping from ξ to the six faces of a die gives a flavor of this technique in the discrete case.

Another example of a continuous random variable is one that ranges over the real numbers between 0 and 2, where the probability of it taking on any particular value x is proportional to the value $2 - x$: it is twice as likely for this random variable to take on

2 Although the theory of Monte Carlo is based on using truly random numbers, in practice a well-written pseudo-random number generator (PRNG) is sufficient. pbrt uses a particularly high-quality PRNG that returns a sequence of pseudo-random values that is effectively as "random" as true random numbers. (Many PRNGs are not as well implemented and have detectable patterns in the sequence of numbers they generate.) True random numbers, found by measuring random phenomena like atomic decay or atmospheric noise, are available from sources like *www.random.org* for those for whom PRNGs are not acceptable.

a value around zero as it is to take one around one, and so forth. The *probability density function* (PDF) formalizes this idea: it describes the relative probability of a random variable taking on a particular value. The PDF $p(x)$ is the derivative of the random variable's CDF,

$$p(x) = \frac{\mathrm{d}P(x)}{\mathrm{d}x}.$$

For uniform random variables, $p(x)$ is a constant; this is a direct consequence of uniformity. For ξ we have

$$p(x) = \begin{cases} 1 & x \in [0, 1) \\ 0 & \text{otherwise.} \end{cases}$$

PDFs are necessarily nonnegative and always integrate to one over their domains. Given an arbitrary interval $[a, b]$ in the domain, the PDF can give the probability that a random variable lies inside the interval:

$$P(x \in [a, b]) = \int_a^b p(x)\, \mathrm{d}x.$$

This follows directly from the first fundamental theorem of calculus and the definition of the PDF.

13.1.2 EXPECTED VALUES AND VARIANCE

The *expected value* $E_p[f(x)]$ of a function f is defined as the average value of the function over some distribution of values $p(x)$ over its domain. In the next section, we will see how Monte Carlo integration computes the expected values of arbitrary integrals. Expected value over a domain, D, is defined as

$$E_p[f(x)] = \int_D f(x)\, p(x)\, \mathrm{d}x. \tag{13.1}$$

As an example, consider the problem of finding the expected value of the cosine function between 0 and π, where p is uniform.[3] Because the PDF $p(x)$ must integrate to one over the domain, $p(x) = 1/\pi$, so

$$E[\cos x] = \int_0^\pi \frac{\cos x}{\pi}\, \mathrm{d}x = \frac{1}{\pi}(-\sin \pi + \sin 0) = 0,$$

which is precisely the expected result. (Consider the graph of $\cos x$ over $[0, \pi]$ to see why this is so.)

The *variance* of a function is the expected deviation of the function from its expected value. Variance is a fundamental concept for quantifying the error in a value estimated by a Monte Carlo algorithm. It provides a precise way to quantify this error and measure how improvements to Monte Carlo algorithms reduce the error in the final result. Most of Chapter 14 is devoted to techniques for reducing variance and thus improving the

3 When computing expected values with a uniform distribution, we will drop the subscript p from E_p.

results computed by `pbrt`. The variance of a function f is defined as

$$V[f(x)] = E\left[\left(f(x) - E[f(x)]\right)^2\right].$$

The expected value and variance have three important properties that follow immediately from their respective definitions:

$$E[af(x)] = aE[f(x)]$$
$$E\left[\sum_i f(X_i)\right] = \sum_i E[f(X_i)]$$
$$V[af(x)] = a^2 V[f(x)].$$

These properties, and some simple algebraic manipulation, yield a much simpler expression for the variance:

$$V[f(x)] = E\left[(f(x))^2\right] - E[f(x)]^2. \quad (13.2)$$

Thus, the variance is the expected value of the square minus the square of the expected value. Given random variables that are *independent*, variance also has the property that the sum of the variances is equal to the variance of their sum:

$$\sum_i V[f(X_i)] = V\left[\sum_i f(X_i)\right].$$

13.2 THE MONTE CARLO ESTIMATOR

We can now define the basic Monte Carlo estimator, which approximates the value of an arbitrary integral. It is the foundation of the light transport algorithms defined in Chapters 15 to 17.

Suppose that we want to evaluate a one-dimensional integral $\int_a^b f(x)\,\mathrm{d}x$. Given a supply of uniform random variables $X_i \in [a, b]$, the Monte Carlo estimator says that the expected value of the estimator

$$F_N = \frac{b-a}{N}\sum_{i=1}^{N} f(X_i),$$

$E[F_N]$, is in fact equal to the integral.[4] This fact can be demonstrated with just a few steps. First, note that the PDF $p(x)$ corresponding to the random variable X_i must be equal to $1/(b-a)$, since p must both be a constant and also integrate to one over the domain $[a, b]$. Algebraic manipulation then shows that

`Lerp()` 1000
`RNG::RandomFloat()` 1003

4 For example, the samples X_i might be computed in an implementation by `Lerp(rng.RandomFloat(), a, b)`.

$$\begin{aligned}
E[F_N] &= E\left[\frac{b-a}{N}\sum_{i=1}^{N} f(X_i)\right] \\
&= \frac{b-a}{N}\sum_{i=1}^{N} E\left[f(X_i)\right] \\
&= \frac{b-a}{N}\sum_{i=1}^{N}\int_a^b f(x)\, p(x)\, \mathrm{d}x \\
&= \frac{1}{N}\sum_{i=1}^{N}\int_a^b f(x)\, \mathrm{d}x \\
&= \int_a^b f(x)\, \mathrm{d}x.
\end{aligned}$$

The restriction to uniform random variables can be relaxed with a small generalization. This is an extremely important step, since carefully choosing the PDF from which samples are drawn is an important technique for reducing variance in Monte Carlo (Section 14.4). If the random variables X_i are drawn from some arbitrary PDF $p(x)$, then the estimator

$$F_N = \frac{1}{N}\sum_{i=1}^{N}\frac{f(X_i)}{p(X_i)} \tag{13.3}$$

can be used to estimate the integral instead. The only limitation on $p(x)$ is that it must be nonzero for all x where $|f(x) > 0|$. It is similarly easy to see that the expected value of this estimator is the desired integral of f:

$$\begin{aligned}
E[F_N] &= E\left[\frac{1}{N}\sum_{i=1}^{N}\frac{f(X_i)}{p(X_i)}\right] \\
&= \frac{1}{N}\sum_{i=1}^{N}\int_a^b \frac{f(x)}{p(x)} p(x)\, \mathrm{d}x \\
&= \frac{1}{N}\sum_{i=1}^{N}\int_a^b f(x)\, \mathrm{d}x \\
&= \int_a^b f(x)\, \mathrm{d}x.
\end{aligned}$$

Extending this estimator to multiple dimensions or complex integration domains is straightforward. N samples X_i are taken from a multidimensional (or "joint") PDF, and the estimator is applied as usual. For example, consider the three-dimensional integral

$$\int_{x_0}^{x_1}\int_{y_0}^{y_1}\int_{z_0}^{z_1} f(x, y, z)\, \mathrm{d}x\, \mathrm{d}y\, \mathrm{d}z.$$

If samples $X_i = (x_i, y_i, z_i)$ are chosen uniformly from the box from (x_0, y_0, z_0) to (x_1, y_1, z_1), the PDF $p(X)$ is the constant value

$$\frac{1}{(x_1 - x_0)} \frac{1}{(y_1 - y_0)} \frac{1}{(z_1 - z_0)},$$

and the estimator is

$$\frac{(x_1 - x_0)(y_1 - y_0)(z_1 - z_0)}{N} \sum_i f(X_i).$$

Note that the number of samples N can be chosen arbitrarily, regardless of the dimension of the integrand. This is another important advantage of Monte Carlo over traditional deterministic quadrature techniques. The number of samples taken in Monte Carlo is completely independent of the dimensionality of the integral, while with standard numerical quadrature techniques the number of samples required is exponential in the dimension.

Showing that the Monte Carlo estimator converges to the right answer is not enough to justify its use; a good rate of convergence is important too. Although we will not derive its rate of convergence here, it has been shown that error in the Monte Carlo estimator decreases at a rate of $O(\sqrt{N})$ in the number of samples taken. An accessible treatment of this topic can be found in Veach's thesis (Veach 1997, p. 39). Although standard quadrature techniques converge faster than $O(\sqrt{N})$ in one dimension, their performance becomes exponentially worse as the dimensionality of the integrand increases, while Monte Carlo's convergence rate is independent of the dimension, making Monte Carlo the only practical numerical integration algorithm for high-dimensional integrals. We have already encountered some high-dimensional integrals in this book, and in Chapter 15 we will see that the path tracing formulation of the light transport equation is an *infinite-dimensional* integral!

13.3 BASIC SAMPLING OF RANDOM VARIABLES

In order to evaluate the Monte Carlo estimator in Equation (13.3), it is necessary to be able to draw random samples from the chosen probability distribution. This section will introduce the basics of this process and demonstrate it with some straightforward examples. The next two sections will introduce more complex approaches to sampling before Section 13.6 develops the approach for the general multidimensional case. Sections 14.5 and 14.6 in the next chapter will show how to use these techniques to generate samples from the distributions defined by BSDFs and light sources.

13.3.1 THE INVERSION METHOD

The inversion method uses one or more uniform random variables and maps them to random variables from the desired distribution. To explain how this process works in general, we will start with a simple discrete example. Suppose we have a process with four possible outcomes. The probabilities of each of the four outcomes are given by p_1, p_2, p_3, and p_4, respectively, with the requirement that $\sum_{i=1}^{4} p_i = 1$. The corresponding PDF is shown in Figure 13.1.

In order to draw a sample from this distribution, we first find the CDF $P(x)$. In the continuous case, P is the indefinite integral of p. In the discrete case, we can directly construct the CDF by stacking the bars on top of each other, starting at the left. This idea

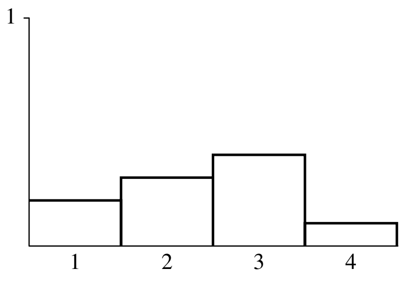


Figure 13.1: A Discrete PDF for Four Events Each with a Probability p_i. The sum of their probabilities $\sum_i p_i$ is necessarily one.

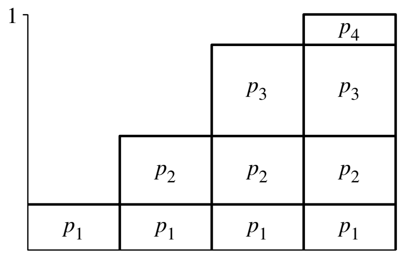


Figure 13.2: A Discrete CDF, Corresponding to the PDF in Figure 13.1. Each column's height is given by the PDF for the event that it represents plus the sum of the PDFs for the previous events, $P_i = \sum_{j=1}^{i} p_i$.

is shown in Figure 13.2. Notice that the height of the rightmost bar must be one because of the requirement that all probabilities sum to one.

To draw a sample from the distribution, we then take a uniform random number ξ and use it to select one of the possible outcomes using the CDF, doing so in a way that chooses a particular outcome with probability equal to its own probability. This idea is illustrated in Figure 13.3, where the events' probabilities are projected onto the vertical axis and a random variable ξ selects among them. It should be clear that this draws from the correct distribution—the probability of the uniform sample hitting any particular bar is exactly equal to the height of that bar. In order to generalize this technique to continuous distributions, consider what happens as the number of discrete possibilities approaches infinity. The PDF from Figure 13.1 becomes a smooth curve, and the CDF from Figure 13.2 becomes its integral. The projection process is still the same, although if the function is continuous, the projection has a convenient mathematical interpretation—it represents inverting the CDF and evaluating the inverse at ξ. This technique is thus called the *inversion method*.

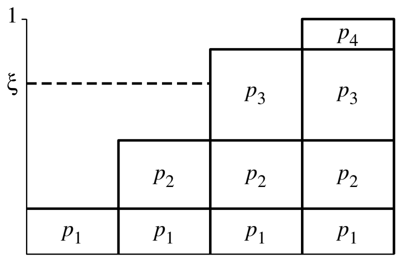


Figure 13.3: To use the inversion method to draw a sample from the distribution described by the PDF in Figure 13.1, a canonical uniform random variable is plotted on the vertical axis. By construction, the horizontal extension of ξ will intersect the box representing the ith outcome with probability p_i. If the corresponding event is chosen for a set of random variables ξ, then the resulting distribution of events will be distributed according to the PDF.

More precisely, we can draw a sample X_i from an arbitrary PDF $p(x)$ with the following steps:

1. Compute the CDF[5] $P(x) = \int_0^x p(x')\,\mathrm{d}x'$.
2. Compute the inverse $P^{-1}(x)$.
3. Obtain a uniformly distributed random number ξ.
4. Compute $X_i = P^{-1}(\xi)$.

Example: Power Distribution

As an example of how this procedure works, consider the task of drawing samples from a *power distribution*, $p(x) \propto x^n$. Sampling from this distribution will be performed when we are trying to sample the Blinn microfacet model. The PDF of the power distribution is

$$p(x) = cx^n,$$

for some constant c. The first task to tackle is to find the PDF for the function. In most cases, this simply involves computing the value of the proportionality constant c, which can be found using the constraint that $\int p(x)\,\mathrm{d}x = 1$:

$$\begin{aligned} \int_0^1 cx^n\,\mathrm{d}x &= 1 \\ c\,\frac{x^{n+1}}{n+1}\bigg|_0^1 &= 1 \\ \frac{c}{n+1} &= 1 \\ c &= n+1. \end{aligned}$$

5 In general, the lower limit of integration should be $-\infty$, although if $p(x) = 0$ for $x < 0$, this equation is equivalent.

Therefore, $p(x) = (n+1)x^n$. We can integrate this to get the CDF:

$$P(x) = \int_0^x p(x)\,\mathrm{d}x = x^{n+1},$$

and inversion is simple: $P^{-1}(x) = \sqrt[n+1]{x}$. Therefore, given a uniform random variable ξ, samples can be drawn from the power distribution as

$$X = \sqrt[n+1]{\xi}. \tag{13.4}$$

Another approach is to use a sampling trick that works only for the power distribution, selecting $X = \max(\xi_1, \xi_2, \ldots, \xi_{n+1})$. This random variable is distributed according to the power distribution as well. To see why, note that $Pr\{X < x\}$ is the probability that *all* the $\xi_i < x$. But the ξ_i are independent, so

$$Pr\{X < x\} = \prod_{i=1}^{n+1} Pr\left\{\xi_i < x\right\} = x^{n+1},$$

which is exactly the desired CDF. Depending on the speed of your random number generator, this technique can be faster than the inversion method for small values of n.

Example: Exponential Distribution

When rendering images with participating media, it is frequently useful to draw samples from a distribution $p(x) \propto \mathrm{e}^{-ax}$. As before, the first step is to normalize this distribution so that it integrates to one. In this case, the range of values x we'd like the generated samples to cover is $[0, \infty)$ rather than $[0, 1]$, so

$$\int_0^\infty c\mathrm{e}^{-ax}\,\mathrm{d}x = -\left.\frac{c}{a}\mathrm{e}^{-ax}\right|_0^\infty = \frac{c}{a} = 1.$$

Thus we know that $c = a$, and our PDF is $p(x) = a\mathrm{e}^{-ax}$. Now, we integrate to find $P(x)$:

$$P(x) = \int_0^x a\mathrm{e}^{-ax'}\,\mathrm{d}x' = 1 - \mathrm{e}^{-ax}.$$

This function is easy to invert:

$$P^{-1}(x) = -\frac{\ln(1-x)}{a},$$

and we can draw samples thusly:

$$X = -\frac{\ln(1-\xi)}{a}.$$

This equation can be further simplified by making the observation that if ξ is a uniformly distributed random number, so is $1-\xi$, so we can replace $1-\xi$ by ξ without changing the distribution. Therefore, our final sampling strategy is

$$X = -\frac{\ln(\xi)}{a}.$$

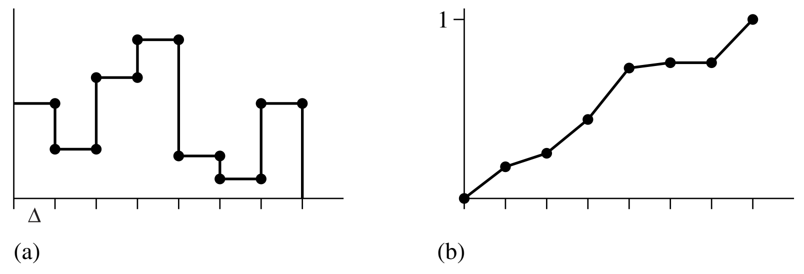


Figure 13.4: (a) Probability density function for a piecewise-constant 1D function and (b) cumulative distribution function defined by this PDF.

Example: Piecewise-Constant 1D Functions

An interesting exercise is to work out how to sample from 1D piecewise-constant functions (step functions). Without loss of generality, we will just consider piecewise-constant functions defined over [0, 1].

Assume that the 1D function's domain is split into N equal-sized pieces of size $\Delta = 1/N$. These regions start and end at points $x_i = i\Delta$, where i ranges from 0 to N, inclusive. Within each region, the value of the function $f(x)$ is a constant (Figure 13.4(a)). The value of $f(x)$ is

$$f(x) = \begin{cases} v_0 & x_0 \le x < x_1 \\ v_1 & x_1 \le x < x_2 \\ \vdots \end{cases} .$$

The integral $\int f(x)\,\mathrm{d}x$ is

$$c = \int_0^1 f(x)\,\mathrm{d}x = \sum_{i=0}^{N-1} \Delta v_i = \sum_{i=0}^{N-1} \frac{v_i}{N}, \tag{13.5}$$

and so it is easy to construct the PDF $p(x)$ for $f(x)$ as $f(x)/c$. By direct application of the relevant formulae, the CDF $P(x)$ is a piecewise linear function defined at points x_i by

$$\begin{aligned} P(x_0) &= 0 \\ P(x_1) &= \int_{x_0}^{x_1} p(x)\,\mathrm{d}x = \frac{v_0}{Nc} = P(x_0) + \frac{v_0}{Nc} \\ P(x_2) &= \int_{x_0}^{x_2} p(x)\,\mathrm{d}x = \int_{x_0}^{x_1} p(x)\,\mathrm{d}x + \int_{x_1}^{x_2} p(x)\,\mathrm{d}x = P(x_1) + \frac{v_1}{Nc} \\ P(x_i) &= P(x_{i-1}) + \frac{v_{i-1}}{Nc}. \end{aligned}$$

Between two points x_i and x_{i+1}, the CDF is linearly increasing with slope v_i/c.

Recall that in order to sample $f(x)$ we need to invert the CDF to find the value x such that

$$\xi = \int_0^x p(x')\,\mathrm{d}x' = P(x).$$

Because the CDF is monotonically increasing, the value of x must be between the x_i and x_{i+1} such that $P(x_i) \le \xi$ and $\xi \le P(x_{i+1})$. Given an array of CDF values, this pair of $P(x_i)$ values can be efficiently found with a binary search.

`Distribution1D` is a small utility class that represents a piecewise-constant 1D function's PDF and CDF and provides methods to perform this sampling efficiently.

```
⟨Monte Carlo Utility Declarations⟩ +≡
    struct Distribution1D {
        ⟨Distribution1D Public Methods 648⟩
    private:
        ⟨Distribution1D Private Data 648⟩
    };
```

The `Distribution1D` constructor takes `n` values of a piecewise-constant function `f`. It makes its own copy of the function values, computes the function's CDF, and also stores the integral of the function, `funcInt`. Note that the constructor allocates `n+1` floats for the `cdf` array because if $f(x)$ has N step values, then we need to store the value of the CDF at each of the $N + 1$ values of x_i. Storing the CDF value of 1 at the end of the array is redundant but simplifies the sampling code later.

```
⟨Distribution1D Public Methods⟩ ≡                                    648
    Distribution1D(const float *f, int n) {
        count = n;
        func = new float[n];
        memcpy(func, f, n*sizeof(float));
        cdf = new float[n+1];
        ⟨Compute integral of step function at x_i 648⟩
        ⟨Transform step function integral into CDF 649⟩
    }
```

```
⟨Distribution1D Private Data⟩ ≡                                      648
    float *func, *cdf;
    float funcInt;
    int count;
```

This constructor computes the integral of $f(x)$ using Equation (13.5). It stores the result in the `cdf` array for now so that it doesn't need to allocate additional temporary space for it.

```
⟨Compute integral of step function at x_i⟩ ≡                         648
    cdf[0] = 0.;
    for (int i = 1; i < count+1; ++i)
        cdf[i] = cdf[i-1] + func[i-1] / n;
```

Distribution1D 648
Distribution1D::cdf 648
Distribution1D::count 648
Distribution1D::func 648

Now that the value of the integral over all of [0, 1] is stored in `cdf[n]`, this value can be copied into `funcInt` and the CDF can be normalized by dividing through by it:

⟨*Transform step function integral into CDF*⟩ ≡ **648**
```
funcInt = cdf[count];
for (int i = 1; i < n+1; ++i)
    cdf[i] /= funcInt;
```

The `Distribution1D::SampleContinuous()` method uses the given random sample `u` to sample from its distribution. It returns the corresponding value $x \in [0, 1)$ and the value of the PDF $p(x)$.

⟨*Distribution1D Public Methods*⟩ +≡ **648**
```
float SampleContinuous(float u, float *pdf) const {
    ⟨Find surrounding CDF segments and offset 649⟩
    ⟨Compute offset along CDF segment 649⟩
    ⟨Compute PDF for sampled offset 649⟩
    ⟨Return x ∈ [0, 1) corresponding to sample 649⟩
}
```

This method first finds the pair of CDF values that straddle `u`, setting `offset` to the offset to the first of the two of them. Because the `cdf` array is monotonically increasing (and is thus sorted), we can use a binary search function provided by the C++ standard library: `lower_bound()` takes a pointer to the start of the array and a pointer one position past the end of the array as well as the value to search for. Explicitly storing the value 1 at the end of the array makes it possible to just use `lower_bound()` directly and not include additional logic to handle the end-case. In the rare (but valid) case of `u == 0`, then the `max()` test below ensures that the value −1 isn't incorrectly computed for `offset`.

⟨*Find surrounding CDF segments and* `offset`⟩ ≡ **649, 650**
```
float *ptr = std::lower_bound(cdf, cdf+count+1, u);
int offset = max(0, int(ptr-cdf-1));
```

Given the pair of CDF values, we can compute x. First, we determine how far `u` is between `cdf[offset]` and `cdf[offset+1]`, `du`, where `du` is zero if `u == cdf[offset]` and goes up to one if `u == cdf[offset+1]`. Because the CDF is piecewise linear, the sample value x is the same offset between x_i and x_{i+1} (Figure 13.4(b)).

⟨*Compute offset along CDF segment*⟩ ≡ **649**
```
float du = (u - cdf[offset]) / (cdf[offset+1] - cdf[offset]);
```

The PDF for this sample $p(x)$ is easily computed since we have the function's integral in `funcInt`.

⟨*Compute PDF for sampled offset*⟩ ≡ **649**
```
if (pdf) *pdf = func[offset] / funcInt;
```

Finally, the appropriate value of x is computed and returned.

⟨*Return $x \in [0, 1)$ corresponding to sample*⟩ ≡ **649**
```
return (offset + du) / count;
```

Distribution1D::cdf 648
Distribution1D::count 648
Distribution1D::func 648
Distribution1D::funcInt 648
Distribution1D::SampleContinuous() 649

In a small overloading of semantics, `Distribution1D` can also be used for discrete 1D probability distributions where there are some number of buckets n, each with some weight, and we'd like to sample among the buckets with probability proportional to their relative weights. This functionality is used, for example, by some of the `Integrators` in Chapter 15 that compute a discrete distribution for the light sources in the scene with weights given by the lights' powers. Sampling from the discrete distribution just requires figuring out which pair of CDF values the sample value lies between; the PDF is computed as the discrete probability of sampling the corresponding bucket.

⟨*Distribution1D Public Methods*⟩ +≡ **648**

```
int SampleDiscrete(float u, float *pdf) const {
    ⟨Find surrounding CDF segments and offset 649⟩
    if (pdf) *pdf = func[offset] / (funcInt * count);
    return offset;
}
```

13.3.2 THE REJECTION METHOD

For some functions $f(x)$, it may not be possible to integrate them in order to find their PDFs, or it may not be possible to analytically invert their CDFs. The *rejection method* is a technique for generating samples according to a function's distribution without needing to do either of these steps; it is essentially a dart-throwing approach. Assume that we want to draw samples from some such function $f(x)$ but we do have a PDF $p(x)$ that satisfies $f(x) < c\ p(x)$ for some scalar constant c, and suppose that we do know how to sample from p. The rejection method is then:

```
loop forever:
  sample $X$ from $p$'s distribution
  if $\xi < f(X)/(c\ p(X))$ then
    return $X$
```

This procedure repeatedly chooses a pair of random variables (X, ξ). If the point $(X, \xi\ c\ p(X))$ lies under $f(X)$, then the sample X is accepted. Otherwise, it is rejected and a new sample pair is chosen. This idea is illustrated in Figure 13.5. Without going into too much detail, it should be clear that the efficiency of this scheme depends on how tightly $c\ p(x)$ bounds $f(x)$. This technique works in any number of dimensions.

In practice, rejection sampling isn't used in any of the Monte Carlo algorithms currently implemented in `pbrt`. We will normally prefer to find distributions that are similar to $f(x)$ that can be sampled directly, for reasons that will be explained in the next chapter. Nevertheless, rejection sampling is an important technique to be aware of, particularly when debugging Monte Carlo implementations. For example, if one suspects the presence of a bug in code that draws samples from some distribution using the inversion method, then one can replace it with a straightforward implementation based on the rejection method and see if the Monte Carlo estimator computes the same result. Of course, it's necessary to take many samples in situations like these, so that variance in the estimates doesn't mask errors.

`Distribution1D` 648
`Distribution1D::count` 648
`Distribution1D::func` 648
`Distribution1D::funcInt` 648

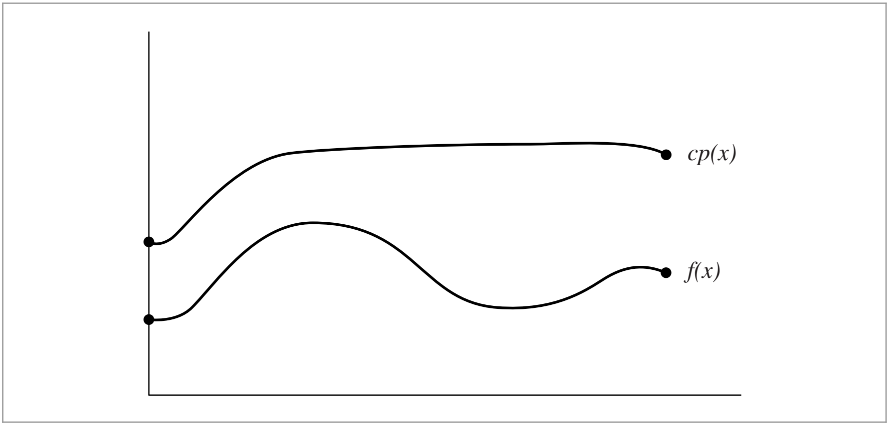


Figure 13.5: Rejection sampling generates samples according to the distribution of an arbitrary function $f(x)$ even if f's PDF is unknown or its CDF can't be inverted. If some distribution $p(x)$ and a scalar constant c are known such that $f(x) < c\,p(x)$, then samples can be drawn from $p(x)$ and randomly accepted with the rejection method. The closer the fit of $c\,p(x)$ to $f(x)$, the more efficient this process is.

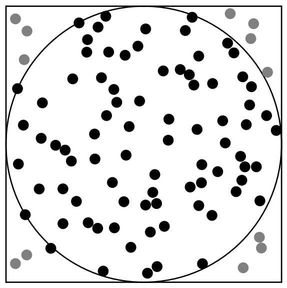


Figure 13.6: Rejection Sampling a Circle. One approach to finding uniform points in the unit circle is to sample uniform random points in the unit square and reject all that lie outside the circle. The remaining points will be uniformly distributed within the circle.

Example: Rejection Sampling a Unit Circle

Suppose we want to select a uniformly distributed point inside a unit circle. Using the rejection method, we simply select a random (x, y) position inside the circumscribed square and return it if it falls inside the circle. This process is shown in Figure 13.6.

`RejectionSampleDisk()` 652

The function `RejectionSampleDisk()` implements this algorithm. A similar approach will work to generate uniformly distributed samples inside for any complex shape as long as it has an inside–outside test.

⟨*Monte Carlo Function Definitions*⟩ +≡
```
void RejectionSampleDisk(float *x, float *y, RNG &rng) {
    float sx, sy;
    do {
        sx = 1.f - 2.f * rng.RandomFloat();
        sy = 1.f - 2.f * rng.RandomFloat();
    } while (sx*sx + sy*sy > 1.f);
    *x = sx;
    *y = sy;
}
```

In general, the efficiency of rejection sampling depends on the percentage of samples that are expected to be rejected. For the problem of finding uniform points in the 2D case, this is easy to compute. It is the area of the circle divided by the area of the square: $\frac{\pi}{4} \approx 78.5\%$. If the method is applied to generate samples in hyperspheres in the general n-dimensional case, however, the volume of an n-dimensional hypersphere actually goes to *zero* as n increases and this approach becomes increasingly inefficient.

★ 13.4 METROPOLIS SAMPLING

Metropolis sampling is a technique that has the remarkable property that it can generate a set of samples from any non-negative function f that are distributed proportionally to f's value (Metropolis et al. 1953).[6] Remarkably, it does this without requiring anything more than the ability to evaluate f; it is not necessary to be able to integrate f, normalize the integral, and invert the resulting PDF. Furthermore, *every* sample generated is from the function's PDF; Metropolis sampling doesn't share the shortcoming of rejection sampling that many candidate samples may be rejected before one is accepted. It can thus efficiently generate samples from a wider variety of functions than the techniques introduced in the previous section. It forms the foundation of the Metropolis light transport algorithm implemented in Section 15.7.

Metropolis sampling does have a few disadvantages: successive samples in the sequence are often correlated, and it is thus not possible to ensure that a small number of samples generated by Metropolis is well distributed across the domain. It's only in the limit over a large number of samples that the samples will cover the domain. As such, the variance reduction advantages of techniques like stratified sampling (Section 14.2.1) are generally not available when using Metropolis sampling.

13.4.1 BASIC ALGORITHM

More concretely, the Metropolis algorithm generates a set of samples X_i from a function f, which is defined over an arbitrary-dimensional state space Ω (frequently, $\Omega = \mathbb{R}^n$) and returns a value in the reals, $f{:}\Omega \rightarrow \mathbb{R}$. After the first sample $X_0 \in \Omega$ has been selected,

RNG 1003
RNG::RandomFloat() 1003

6 We will refer to the Monte Carlo sampling algorithm as "the Metropolis algorithm" here. Other commonly used shorthands for it include M(RT)^2, for the initials of the authors of the original paper, and Metropolis–Hastings, which gives a nod to Hastings, who generalized the technique (Fishman 1996). It is also commonly known as Markov Chain Monte Carlo.

each subsequent sample X_i is generated by using a random *mutation* to X_{i-1} to compute a proposed sample X'. The mutation may be accepted or rejected, and X_i is accordingly set to either X' or X_{i-1}. When these transitions from one state to another are chosen subject to a few requirements (to be described shortly), the distribution of X_i values that results reaches an equlibrium distribution; this distribution is the *stationary distribution*. In the limit, the distribution of the set of samples $X_i \in \Omega$ is proportional to $f(x)$'s probability density function $p(x) = f(x)/\int_\Omega f(x)\mathrm{d}\Omega$.

In order to generate the correct distribution of samples, it is necessary to generate proposed mutations and then accept or reject the mutations subject to a few constraints. Assume that we have a method of proposing mutations that proposes changing from a given state X into a proposed state X' (this might be done by perturbing X in some way, or even by generating a completely new value). We must be able to compute a tentative transition function $T(X \to X')$ that gives the probability density of the mutation technique's proposing a transition to X', given that the current state is X. (Section 13.4.2 will discuss considerations for designing transition functions.)

Given a transition function, it is possible to define an *acceptance probability* $a(X \to X')$ that gives the probability of accepting a proposed mutation from X to X' in a way that ensures that the distribution of samples is proportional to $f(x)$. If the distribution is already in equlibrium, the transition density between any two states must be equal:[7]

$$f(X)\,T(X \to X')\,a(X \to X') = f(X')\,T(X' \to X)\,a(X' \to X). \qquad [13.6]$$

This property is called *detailed balance*.

Since f and T are set, Equation (13.6) tells us how a must be defined. In particular, a definition of a that maximizes the rate at which equlibrium is reached is

$$a(X \to X') = \min\left(1, \frac{f(X')\,T(X' \to X)}{f(X)\,T(X \to X')}\right). \qquad [13.7]$$

One thing to immediately notice from Equation (13.7) is that, if the transition probability density is the same in both directions, the acceptance probability simplifies to

$$a(X \to X') = \min\left(1, \frac{f(X')}{f(X)}\right). \qquad [13.8]$$

Put together, we have the basic Metropolis sampling algorithm in pseudocode:

```
X = X0
for i = 1 to n
    X' = mutate(X)
    a = accept(X, X')
    if (random() < a)
        X = X'
    record(X)
```

7 See Kalos and Whitlock (1986) or Veach's thesis (1997) for a rigorous derivation.

This code generates n samples by mutating the previous sample and computing acceptance probabilities as in Equation (13.7). Each sample X_i can then be recorded in a data structure or used as a sample for integration.

Because the Metropolis algorithm naturally avoids parts of Ω where $f(x)$'s value is relatively low, few samples will be accumulated there. In order to get some information about $f(x)$'s behavior in such regions, the *expected values* technique can be used to enhance the basic Metropolis algorithm. In this case, we still decide which state to transition into as before, but we record a sample at each of X and X', regardless of which one is selected by the acceptance criteria. Each of these recorded samples has a weight associated with it, where the weights are the probabilities $(1-a)$ for X and a for X', where a is the acceptance probability. Expected values doesn't change the way we decide which state, X or X', to use at the next step; that part of the computation remains the same.

Updated pseudocode shows the idea:

```
X = X0
for i = 1 to n
    X' = mutate(X)
    a = accept(X, X')
    record(X, (1-a) * weight)
    record(X', a * weight)
    if (random() < a)
        X = X'
```

Comparing the two pieces of pseudocode, we can see that in the limit, the same weight distribution will be accumulated for X and X'. Expected values more quickly give a smoother result and more information about the areas where $f(x)$ is low than the basic algorithm does.

13.4.2 CHOOSING MUTATION STRATEGIES

In general, one has a lot of freedom in choosing mutation strategies, subject to being able to compute the tenative transition density $T(X \rightarrow X')$. Recall from Equation (13.8) that if the transition densities are symmetric then it is not even necessary to be able to compute them to apply the Metropolis sampling algorithm. It's easy to apply multiple mutation strategies, so if there are some that are effective in only some circumstances it doesn't hurt to try using them as one of a set of approaches.

It is generally desirable that mutations propose large changes to the current sample rather than small ones. Doing so more quickly explores the state space, rather than letting the sampler get stuck in a small region of it. However, when the function's value $f(X)$ is relatively large at the current sample X, then it is likely that many proposed mutations will be rejected (consider the case where $f(X) \gg f(X')$ in Equation (13.8); $a(X \rightarrow X')$ will be very small). We'd like to avoid the case where many samples in a row are the same, again to better explore new parts of the state space: staying in the same state for many samples in a row leads to increased variance—intuitively, it makes sense that the more we move around Ω, the better the overall results will be. For this case, small mutations are likely to propose samples X' where f is still relatively large, leading to higher acceptance properties.

Thus, one useful mutation approach is to apply random perturbations to the current sample X. If the sample X is a vector of real numbers $(x_0, x_1, \ldots)$, then some or all of the samples x_i can be perturbed. One possibility is to perturb by adding or subtracting a scaled random variable:

$$x_i' = x_i \pm s\,\xi$$

for some scale factor s. Another useful technique is to perturb using values from an exponential distribution:

$$x_i' = x_i \pm b\, \mathrm{e}^{-\log(b/a)\,\xi}, \tag{13.9}$$

which generates an exponentially distributed sample in the range $[a, b]$. This approach can be more robust than using a single scale factor s by frequently proposing small mutations but periodically proposing very large ones. Note that this method is symmetric, so we don't need to compute the transition densities $T(X \rightarrow X')$ when using it with Metropolis sampling.

A related mutation approach is to just discard the current sample entirely and generate a new one with uniform random numbers:

$$x_i = \xi.$$

(Note that this is also a symmetric method.) Occasionally generating a completely new sample in this manner is important since it ensures that we don't get stuck in one part of the state space and never sample the rest of it. In general, it's necessary that it be possible to reach all states $X \in \Omega$ where $f(X) > 0$ with nonzero probability (this property is called *ergodicity*). In particular, to ensure ergodicity it suffices that $T(X \rightarrow X') > 0$ for all X and X' where $f(X) > 0$ and $f(X') > 0$.

Another approach is to use PDFs that match some part of the function being sampled. If we have a PDF $p(x)$ that is similar to some component of f, then we can use that to derive a mutation strategy by just drawing new samples $X \sim p$. In this case, the transition function is straightforward:

$$T(X \rightarrow X') = p(X').$$

In other words, the current state X doesn't matter for computing the transition density: we propose a transition into a state X' with a density that depends only on the newly proposed state X' and not at all on the current state.

13.4.3 START-UP BIAS

One issue that we've sidestepped thus far is how the initial sample X_0 is computed. The transition and acceptance methods above tell us how to generate new samples X_{i+1}, but all presuppose that the current sample X_i has itself *already* been sampled with probability proportional to f. Using a sample not from f's distribution leads to a problem called *start-up bias*.

A common solution to this problem is to run the Metropolis sampling algorithm for some number of iterations from an arbitrary starting state, discard the samples that are generated, and then start the process for real, assuming that that has brought us to an appropriately sampled X value. This is unsatisfying for two reasons: first, the expense of taking the samples that were then discarded may be high, and, second, we can only guess at how many initial samples must be taken in order to remove start-up bias.

A straightforward alternative can be used if another sampling method is available: an initial value X_0 is sampled using any density function $X_0 \sim p(x)$. We start the Markov chain from the state X_0, but we weight the contributions of all of the samples that we generate by the weight

$$w = \frac{f(X_0)}{p(X_0)}.$$

This method eliminates start-up bias completely and does so in a predictable manner.

The only potential problem comes if $f(X_0) = 0$ for the X_0 we chose; in this case, all samples will have a weight of zero! This doesn't mean that the algorithm is biased, however; the expected value of the result still converges to the correct distribution (see Veach (1997) for further discussion and for a proof of the correctness). To reduce variance and avoid this risk, we can instead sample a set of N candidate sample values, $Y_1, \ldots, Y_N$, defining a weight for each by

$$w_i = \frac{f(Y_i)}{p(Y_i)}.$$

We then choose the starting X_0 sample for the Metropolis algorithm from the Y_i with probability proportional to their relative weights and compute a sample weight w as the average of all of the w_i weights. All subsequent samples X_i that are generated by the Metropolis algorithm are then weighted by the sample weight w.

13.4.4 ESTIMATING INTEGRALS WITH METROPOLIS SAMPLING

We can apply the Metropolis algorithm to estimating integrals such as $\int f(x)g(x)\,\mathrm{d}\Omega$. Doing so is the basis of the Metropolis light transport algorithm implemented in Section 15.7.

To see how the samples from Metropolis sampling can be used in this way, recall that the standard Monte Carlo estimator, Equation (13.3), says that

$$\int_\Omega f(x)g(x)\,\mathrm{d}\Omega \approx \frac{1}{N}\sum_{i=1}^{N}\frac{f(X_i)g(X_i)}{p(X_i)},$$

where X_i are sampled from a density function $p(x)$. Thus, if we apply Metropolis sampling and generate a set of samples, $X_1, \ldots, X_N$, from a density function that is proportional to $f(x)$, then we can estimate this integral as

$$\int_\Omega f(x)g(x)\,\mathrm{d}\Omega \approx \left[\frac{1}{N}\sum_{i=1}^{N} g(X_i)\right]\cdot\int_\Omega f(x)\,\mathrm{d}\Omega. \qquad [13.10]$$

13.4.5 EXAMPLE: ONE-DIMENSIONAL SETTING

In order to illustrate some of the ideas in this section, we'll show how Metropolis sampling can be used to sample a simple one-dimensional function, defined over $\Omega = [0, 1]$

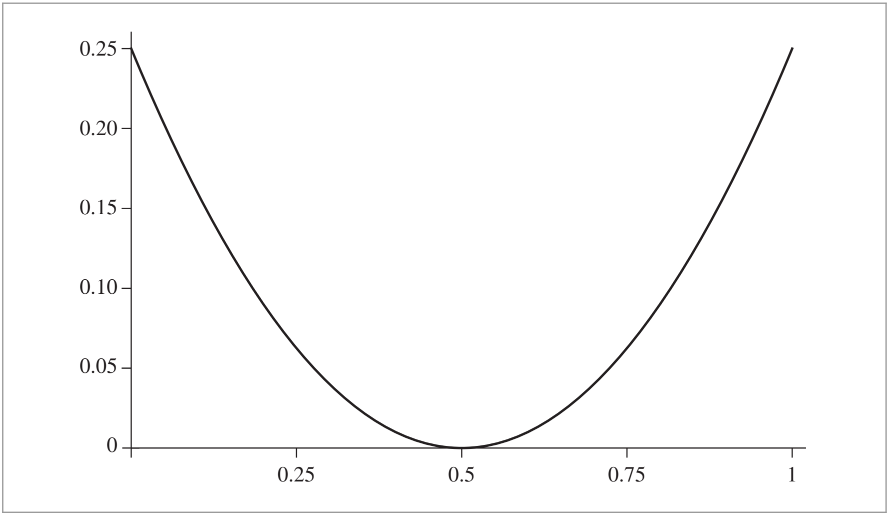


Figure 13.7: Graph of the function $f(x)$ used to illustrate Metropolis sampling in this section.

and zero everywhere else (see Figure 13.7).

$$f(x) = \begin{cases} (x - .5)^2 & 0 \le x \le 1 \\ 0 & \text{otherwise.} \end{cases} \tag{13.11}$$

For this example, assume that we don't actually know the exact form of f—it's just a black box that we can evaluate at particular x values. (Clearly, if we knew that f was just Equation (13.11), there'd be no need for Metropolis in order to draw samples from its distribution!)

We'll define two mutation strategies based on the ideas introduced in Section 13.4.2 and randomly choose among them each time a mutation is proposed, according to a desired distribution of how frequently each is to be used.

The first strategy, mutate$_1$, discards the current sample X and uniformly samples a new one, X', from the entire state space [0, 1]. The transition function for this mutation is straightforward. For mutate$_1$, since we are uniformly sampling over [0, 1], the probability density is uniform over the entire domain; in this case, the density is just one everywhere. We have

$$\text{mutate}_1(X) \rightarrow \xi$$
$$T_1(X \rightarrow X') = 1.$$

The second mutation adds a random offset between $\pm.05$ to the current sample X in an effort to sample repeatedly in the parts of f that make a high contribution to the overall distribution. The transition probability density is zero if X and X' are far enough away that mutate$_2$ will never mutate from one to the other; otherwise, the density is constant. Normalizing the density so that it integrates to one over its domain gives the value 1/0.1. Both this and mutate$_1$ are symmetric, so the transition densities aren't needed to implement the sampling algorithm.

$$\text{mutate}_2(X) \to X + .1(\xi - .5)$$
$$T_2(X \to X') = \begin{cases} \frac{1}{0.1} & |X - X'| \le .05 \\ 0 & \text{otherwise.} \end{cases}$$

To find the initial sample, we only need to take a single sample with a uniform PDF over Ω, since $f(x) > 0$ except for a single point in Ω for which there is zero probability of sampling:

$$X_0 = \xi.$$

The sample weight w is then just $f(X_0)$.

We can now run the Metropolis algorithm and generate samples X_i of f. At each transition, we have two weighted samples to record (recall the expected values pseudocode from Section 13.4.1). A simple approach for reconstructing the approximation to f's probability distribution is to store sums of the weights in a set of buckets of uniform width; each sample falls in a single bucket and contributes to it. Figure 13.8 shows some results. For both graphs, a chain of 10,000 mutations was followed, with the sample weights accumulated in 50 buckets over [0, 1].

On the left graph, only mutate$_1$ was used. This alone isn't a very effective mutation, since it doesn't take advantage of the times when it has found a sample in a region of Ω where f has a relatively large value to generate additional samples in that neighborhood. However, the graph does suggest that the algorithm is converging to the correct distribution.

On the right, one of mutate$_1$ and mutate$_2$ was randomly chosen, with probabilities of 10% and 90%, respectively. We see that for the same number of samples taken, we converge to f's distribution with less variance. This is because the algorithm is more effectively able to concentrate its work in areas where f's value is large, proposing fewer mutations to parts of state space where f's value is low. For example, if $X = .8$ and the second mutation proposes $X' = .75$, this will be accepted $f(.75)/f(.8) \approx 69\%$ of the time, while mutations from .75 to .8 will be accepted $\min(1, 1.44) = 100\%$ of the time. Thus, we see how the algorithm naturally tends to try to avoid spending time sampling around the dip in the middle of the curve.

One important thing to note about these graphs is that the y axis has units that are different than those in Figure 13.7, where f is graphed. Recall that Metropolis sampling provides a set of samples distributed according to f's probability density; as such (for example), we would get the same sample distribution for another function $g = 2f$. If we wish to reconstruct an approximation to f directly from Metropolis samples, we must compute a normalization factor and use it to scale the PDF.

Figure 13.9 shows the surprising result of only using mutate$_2$ to propose sample values. On the left, 10,000 samples were taken using just that mutation. Clearly, things have gone awry—*no* samples $X_i > .5$ were generated and the result doesn't bear much resemblance to f.

Thinking about the acceptance probability again, we can see that it would take a large number of mutations, each with low probability of acceptance, to move X_i down close

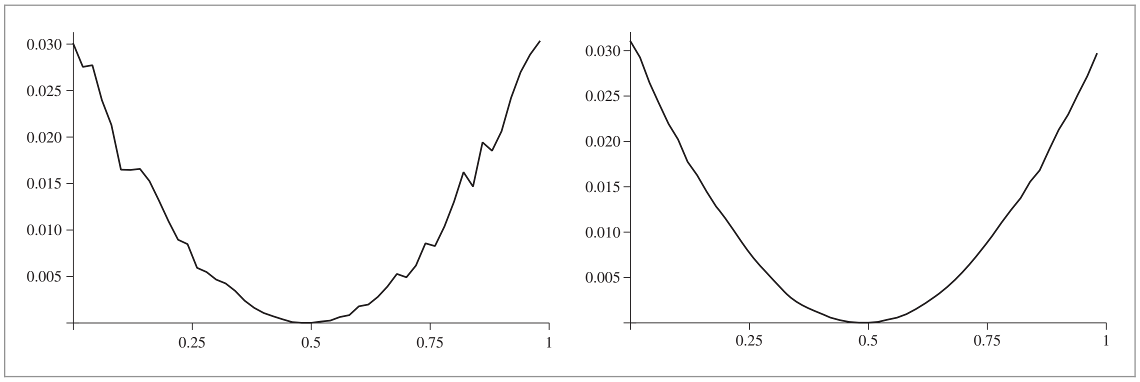


Figure 13.8: Comparison of Metropolis Sampling Strategies. (a) Only mutate_1 was used, randomly selecting a completely new value at each step. (b) Both mutate_1 and mutate_2 were used, with a 1:9 ratio. For the same number of samples, variance is substantially lower, thanks to a higher rate of acceptance of mutate_2's proposed mutations.

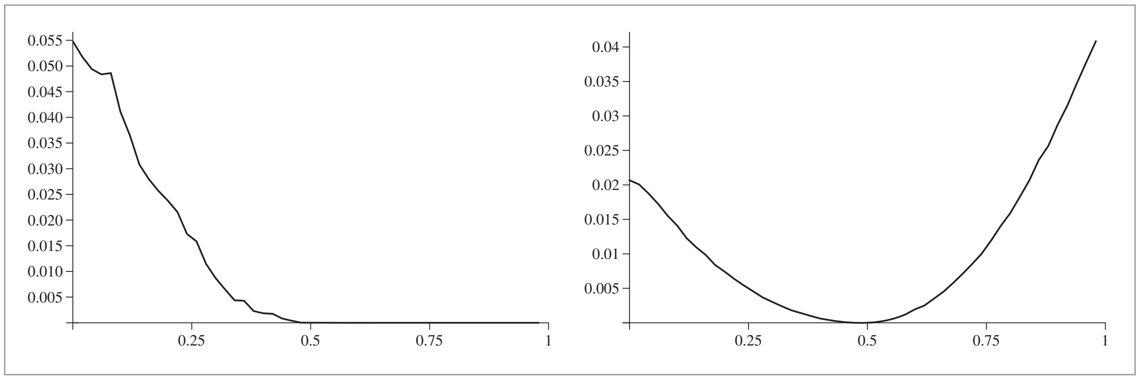


Figure 13.9: Why it is important to periodically pick a completely new sample value with Metropolis sampling. (a) 10,000 iterations using only mutate_2 were computed, (b) 300,000 iterations. It is very unlikely that a series of mutations will be able to move from one side of the curve, across .5, to the other side, since mutations that move toward areas where f's value is low will usually be rejected. As such, the results are inaccurate for these numbers of iterations. (It's small solace that they would be correct in the limit.)

enough to .5 such that mutate_2's short span would be enough to move over to the other side. Since the Metropolis algorithm tends to stay away from the lower-valued regions of f (recall the comparison of probabilities for moving from .8 to .75 versus moving from .75 to .8), this happens quite rarely. The right side of Figure 13.9 shows what happens if 300,000 samples are taken. This was enough to be able to jump from one side of .5 to the other a few times, but not enough to get close to the correct distribution. Using mutate_2 alone is thus not mathematically incorrect, just inefficient: it does have nonzero probability of proposing a transition from any state to any other state (through a chain of multiple transitions).

13.5 TRANSFORMING BETWEEN DISTRIBUTIONS

In describing the inversion method, we introduced a technique that generates samples according to some distribution by transforming canonical uniform random variables in a particular manner. Here, we will investigate the more general question of which distribution results when we transform samples from an arbitrary distribution to some other distribution with a function f.

Suppose we are given random variables X_i that are already drawn from some PDF $p_x(x)$. Now, if we compute $Y_i = y(X_i)$, we would like to find the distribution of the new random variable Y_i. This may seem like an esoteric problem, but we will see that understanding this kind of transformation is critical for drawing samples from multidimensional distribution functions.

The function $y(x)$ must be a one-to-one transformation; if multiple values of x mapped to the same y value, then it would be impossible to unambiguously describe the probability density of a particular y value. A direct consequence of y being one-to-one is that its derivative must either be strictly greater than zero or strictly less than zero, which implies that

$$Pr\{Y \le y(x)\} = Pr\{X \le x\},$$

and therefore

$$P_y(y) = P_y(y(x)) = P_x(x).$$

This relationship between CDFs leads directly to the relationship between their PDFs. If we assume that y's derivative is greater than zero, differentiating gives

$$p_y(y)\frac{\mathrm{d}y}{\mathrm{d}x} = p_x(x),$$

and so

$$p_y(y) = \left(\frac{\mathrm{d}y}{\mathrm{d}x}\right)^{-1} p_x(x).$$

In general, y's derivative is either strictly positive or strictly negative, and the relationship between the densities is

$$p_y(y) = \left|\frac{\mathrm{d}y}{\mathrm{d}x}\right|^{-1} p_x(x).$$

How can we use this formula? Suppose that $p_x(x) = 2x$ over the domain [0, 1], and let $Y = \sin X$. What is the PDF of the random variable Y? Because we know that $\mathrm{d}y/\mathrm{d}x = \cos x$,

$$p_y(y) = \frac{p_x(x)}{|\cos x|} = \frac{2x}{\cos x} = \frac{2 \arcsin y}{\sqrt{1-y^2}}.$$

This procedure may seem backwards—usually we have some PDF that we want to sample from, not a given transformation. For example, we might have X drawn from some

$p_x(x)$ and would like to compute Y from some distribution $p_y(y)$. What transformation should we use? All we need is for the CDFs to be equal, or $P_y(y) = P_x(x)$, which immediately gives the transformation

$$y(x) = P_y^{-1}\left(P_x(x)\right).$$

This is a generalization of the inversion method, since if X were uniformly distributed over $[0, 1]$ then $P_x(x) = x$, and we have the same procedure as was introduced previously.

13.5.1 TRANSFORMATION IN MULTIPLE DIMENSIONS

In the general n-dimensional case, a similar derivation gives the analogous relationship between different densities. We will not show the derivation here; it follows the same form as the one-dimensional case. Suppose we have an n-dimensional random variable X with density function $p_x(x)$. Now let $Y = T(X)$, where T is a bijection. In this case, the densities are related by

$$p_y(y) = p_y(T(x)) = \frac{p_x(x)}{|J_T(x)|},$$

where $|J_T|$ is the absolute value of the determinant of T's Jacobian matrix, which is

$$\begin{pmatrix} \partial T_1/\partial x_1 & \cdots & \partial T_1/\partial x_n \\ \vdots & \ddots & \vdots \\ \partial T_n/\partial x_1 & \cdots & \partial T_n/\partial x_n \end{pmatrix},$$

where T_i are defined by $T(x) = (T_1(x), \ldots, T_n(x))$.

13.5.2 EXAMPLE: POLAR COORDINATES

The polar transformation is given by

$$x = r\cos\theta$$
$$y = r\sin\theta.$$

Suppose we draw samples from some density $p(r, \theta)$. What is the corresponding density $p(x, y)$? The Jacobian of this transformation is

$$J_T = \begin{pmatrix} \frac{\partial x}{\partial r} & \frac{\partial x}{\partial \theta} \\ \frac{\partial y}{\partial r} & \frac{\partial y}{\partial \theta} \end{pmatrix} = \begin{pmatrix} \cos\theta & -r\sin\theta \\ \sin\theta & r\cos\theta \end{pmatrix},$$

and the determinant is $r\left(\cos^2\theta + \sin^2\theta\right) = r$. So $p(x, y) = p(r, \theta)/r$. Of course, this is backwards from what we usually want—typically we start with a sampling strategy in Cartesian coordinates and want to transform it to one in polar coordinates. In that case, we would have

$$p(r, \theta) = r\, p(x, y).$$

13.5.3 EXAMPLE: SPHERICAL COORDINATES

Given the spherical coordinate representation of directions,

$$x = r\sin\theta\cos\phi$$
$$y = r\sin\theta\sin\phi$$
$$z = r\cos\theta,$$

the Jacobian of this transformation has determinant $|J_T| = r^2 \sin\theta$, so the corresponding density function is

$$p(r, \theta, \phi) = r^2 \sin\theta \; p(x, y, z).$$

This transformation is important since it helps us represent directions as points (x, y, z) on the unit sphere. Remember that solid angle is defined as the area of a set of points on the unit sphere. In spherical coordinates, we previously derived

$$d\omega = \sin\theta \; \mathrm{d}\theta \, \mathrm{d}\phi.$$

So if we have a density function defined over a solid angle Ω, this means that

$$Pr\left\{\omega \in \Omega\right\} = \int_\Omega p(\omega)\, \mathrm{d}\omega.$$

The density with respect to θ and ϕ can therefore be derived:

$$\begin{aligned} p(\theta, \phi)\, \mathrm{d}\theta\, \mathrm{d}\phi &= p(\omega)\, \mathrm{d}\omega \\ p(\theta, \phi) &= \sin\theta \; p(\omega). \end{aligned}$$

13.6 2D SAMPLING WITH MULTIDIMENSIONAL TRANSFORMATIONS

Suppose we have a 2D joint density function $p(x, y)$ that we wish to draw samples (X, Y) from. Sometimes multidimensional densities are separable and can be expressed as the product of 1D densities, for example,

$$p(x, y) = p_x(x) p_y(y),$$

for some p_x and p_y. In this case, random variables (X, Y) can be found by independently sampling X from p_x and Y from p_y. Many useful densities aren't separable, however, so we will introduce the theory of how to sample from multidimensional distributions in the general case.

Given a 2D density function, the *marginal density function* $p(x)$ is obtained by "integrating out" one of the dimensions:

$$p(x) = \int p(x, y)\, \mathrm{d}y. \tag{13.12}$$

This can be thought of as the density function for X alone. More precisely, it is the average density for a particular x over *all* possible y values.

The *conditional density function* $p(y|x)$ is the density function for y given that some particular x has been chosen (it is read "p of y given x"):

$$p(y|x) = \frac{p(x, y)}{p(x)}. \tag{13.13}$$

The basic idea for 2D sampling from joint distributions is to first compute the marginal density to isolate one particular variable and draw a sample from that density using standard 1D techniques. Once that sample is drawn, one can then compute the conditional density function given that value and draw a sample from that distribution, again using standard 1D sampling techniques.

13.6.1 EXAMPLE: UNIFORMLY SAMPLING A HEMISPHERE

As an example, consider the task of choosing a direction on the hemisphere uniformly with respect to solid angle. Remember that a uniform distribution means that the density function is a constant, so we know that $p(\omega) = c$. In conjunction with the fact that the density function must integrate to one over its domain, we have

$$\int_{\mathcal{H}^2} p(\omega)\, \mathrm{d}\omega = 1 \Rightarrow c \int_{\mathcal{H}^2} \mathrm{d}\omega = 1 \Rightarrow c = \frac{1}{2\pi}.$$

This tells us that $p(\omega) = 1/(2\pi)$, or $p(\theta, \phi) = \sin\theta/(2\pi)$ (using a result from the previous example about spherical coordinates). Note that this density function is separable. Nevertheless, we will use the marginal and conditional densities to illustrate the multidimensional sampling technique.

Consider sampling θ first. To do so, we need θ's marginal density function $p(\theta)$:

$$p(\theta) = \int_0^{2\pi} p(\theta, \phi)\, \mathrm{d}\phi = \int_0^{2\pi} \frac{\sin\theta}{2\pi}\, \mathrm{d}\phi = \sin\theta.$$

Now, compute the conditional density for ϕ:

$$p(\phi|\theta) = \frac{p(\theta, \phi)}{p(\theta)} = \frac{1}{2\pi}.$$

Notice that the density function for ϕ is itself uniform; this should make intuitive sense given the symmetry of the hemisphere. Now, we use the 1D inversion technique to sample each of these PDFs in turn:

$$P(\theta) = \int_0^{\theta} \sin\theta'\, \mathrm{d}\theta' = 1 - \cos\theta$$
$$P(\phi|\theta) = \int_0^{\phi} \frac{1}{2\pi}\, \mathrm{d}\phi' = \frac{\phi}{2\pi}.$$

Inverting these functions is straightforward, and again noting that we can safely replace $1 - \xi$ with ξ since these are both uniformly distributed random numbers over [0, 1], we get

$$\theta = \cos^{-1}\xi_1$$
$$\phi = 2\pi\xi_2.$$

Converting these back to Cartesian coordinates, we get the final sampling formulae:

$$x = \sin\theta\cos\phi = \cos\left(2\pi\xi_2\right)\sqrt{1 - \xi_1^2}$$
$$y = \sin\theta\sin\phi = \sin\left(2\pi\xi_2\right)\sqrt{1 - \xi_1^2}$$
$$z = \cos\theta = \xi_1.$$

This sampling strategy is implemented in the following code. Two uniform random numbers u1 and u2 are passed in, and a vector on the hemisphere is returned.

⟨*Monte Carlo Function Definitions*⟩ +≡
```
Vector UniformSampleHemisphere(float u1, float u2) {
    float z = u1;
    float r = sqrtf(max(0.f, 1.f - z*z));
    float phi = 2 * M_PI * u2;
    float x = r * cosf(phi);
    float y = r * sinf(phi);
    return Vector(x, y, z);
}
```

For each sampling routine like this in pbrt, there is a corresponding function that returns the value of the PDF for a particular sample. For such functions, it is important to be clear which PDF is being evaluated—for example, for a direction on the hemisphere, we have already seen these densities expressed differently in terms of solid angle and in terms of (θ, ϕ). For hemispheres (and all other directional sampling), these functions return values with respect to solid angle. For the hemisphere, the solid angle PDF is a constant $p(\omega) = 1/(2\pi)$.

⟨*Monte Carlo Function Definitions*⟩ +≡
```
float UniformHemispherePdf() {
    return INV_TWOPI;
}
```

Sampling the full sphere uniformly over its area follows almost exactly the same derivation, which we omit here. The end result is

$$\begin{aligned} x &= \cos(2\pi\xi_2)\sqrt{1-z^2} = \cos(2\pi\xi_2)2\sqrt{\xi_1(1-\xi_1)} \\ y &= \sin(2\pi\xi_2)\sqrt{1-z^2} = \sin(2\pi\xi_2)2\sqrt{\xi_1(1-\xi_1)} \\ z &= 1-2\xi_1. \end{aligned}$$

⟨*Monte Carlo Function Definitions*⟩ +≡
```
Vector UniformSampleSphere(float u1, float u2) {
    float z = 1.f - 2.f * u1;
    float r = sqrtf(max(0.f, 1.f - z*z));
    float phi = 2.f * M_PI * u2;
    float x = r * cosf(phi);
    float y = r * sinf(phi);
    return Vector(x, y, z);
}
```

⟨*Monte Carlo Function Definitions*⟩ +≡
```
float UniformSpherePdf() {
    return 1.f / (4.f * M_PI);
}
```

INV_TWOPI 1002
M_PI 1002
Vector 57

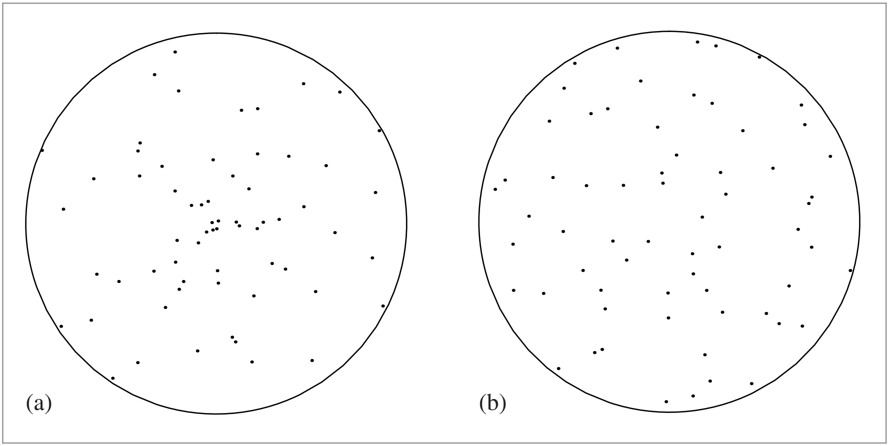


Figure 13.10: (a) When the obvious but incorrect mapping of uniform random variables to points on the disk is used, the resulting distribution is not uniform and the samples are more likely to be near the center of the disk. (b) The correct mapping gives a uniform distribution of points.

13.6.2 EXAMPLE: SAMPLING A UNIT DISK

Although the disk seems a simpler shape to sample than the hemisphere, it can be more tricky to sample uniformly because it has an incorrect intuitive solution. The wrong approach is the seemingly obvious one: $r = \xi_1$, $\theta = 2\pi \xi_2$. Although the resulting point is both random and inside the circle, it is *not* uniformly distributed; it actually clumps samples near the center of the circle. Figure 13.10(a) shows a plot of samples on the unit disk when this mapping was used for a set of uniform random samples (ξ_1, ξ_2). Figure 13.10(b) shows uniformly distributed samples resulting from the following correct approach.

Since we're going to sample uniformly with respect to area, the PDF $p(x, y)$ must be a constant. By the normalization constraint, $p(x, y) = 1/\pi$. If we transform into polar coordinates (see the example in Section 13.5.2), we have $p(r, \theta) = r/\pi$. Now we compute the marginal and conditional densities as before:

$$p(r) = \int_0^{2\pi} p(r, \theta)\, d\theta = 2r$$
$$p(\theta|r) = \frac{p(r, \theta)}{p(r)} = \frac{1}{2\pi}.$$

As with the hemisphere case, the fact that $p(\theta|r)$ is a constant should make sense because of the symmetry of the circle. Integrating and inverting to find $P(r)$, $P^{-1}(r)$, $P(\theta)$, and $P^{-1}(\theta)$, we can find that the correct solution to generate uniformly distributed samples on a disk is

$$r = \sqrt{\xi_1}$$
$$\theta = 2\pi \xi_2.$$

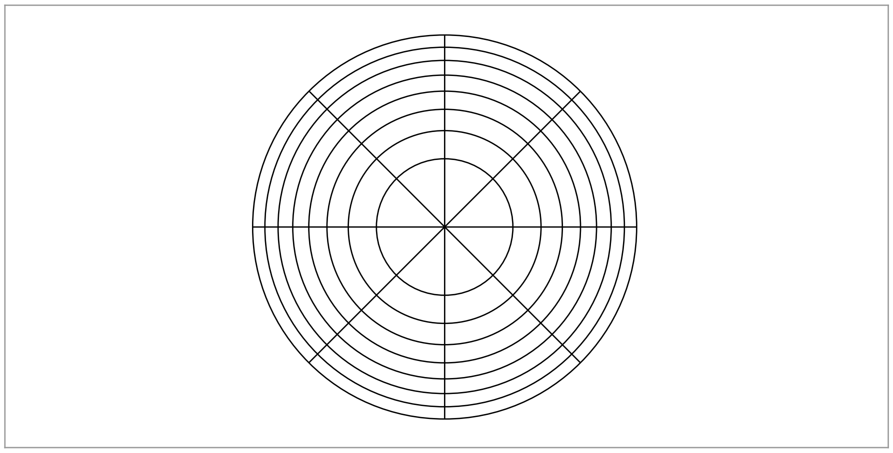


Figure 13.11: The mapping from 2D random samples to points on the disk implemented in `UniformSampleDisk()` distorts areas substantially. Each section of the disk here has equal area and represents $\frac{1}{8}$ of the unit square of uniform random samples in each direction. In general, we'd prefer a mapping that did a better job at mapping nearby (ξ_1, ξ_2) values to nearby points on the disk.

Taking the square root of ξ_1 effectively pushes the samples back toward the edge of the disk, counteracting the clumping referred to earlier.

⟨*Monte Carlo Function Definitions*⟩ +≡
```
void UniformSampleDisk(float u1, float u2, float *x, float *y) {
    float r = sqrtf(u1);
    float theta = 2.0f * M_PI * u2;
    *x = r * cosf(theta);
    *y = r * sinf(theta);
}
```

Although this mapping solves the problem at hand, it distorts areas on the disk; areas on the unit square are elongated and/or compressed when mapped to the disk (Figure 13.11). (Section 14.2.3 will discuss in more detail why this distortion is a disadvantage.) Peter Shirley (1997) developed a "concentric" mapping from the unit square to the unit circle that avoids this problem. The concentric mapping takes points in the square $[-1, 1]^2$ to the unit disk by uniformly mapping concentric squares to concentric circles (Figure 13.12).

The mapping turns wedges of the square into slices of the disk. For example, points in the shaded area of the square in Figure 13.12 are mapped to (r, θ) by

$$r = x$$
$$\theta = \frac{y}{x}.$$

See Figure 13.13. The other four quadrants are handled analogously.

M_PI 1002
UniformSampleDisk() 666

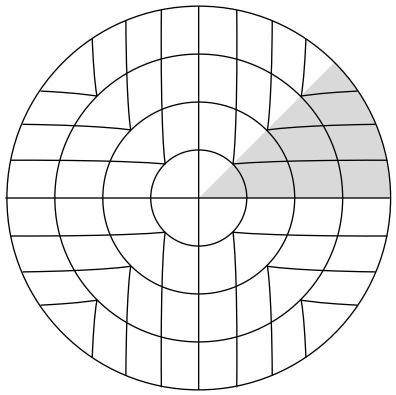


Figure 13.12: The concentric mapping maps squares to circles, giving a less distorted mapping than the first method shown for uniformly sampling points on the unit disk.

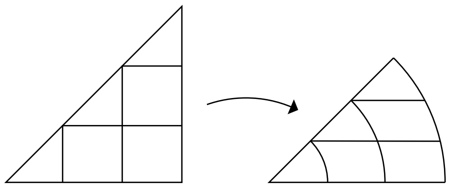


Figure 13.13: Triangular wedges of the square are mapped into (r, θ) pairs in pie-shaped slices of the circle.

⟨*Monte Carlo Function Definitions*⟩ $+\equiv$

```
void ConcentricSampleDisk(float u1, float u2, float *dx, float *dy) {
    float r, theta;
    ⟨Map uniform random numbers to [−1, 1]² 667⟩
    ⟨Map square to (r, θ) 668⟩
    *dx = r * cosf(theta);
    *dy = r * sinf(theta);
}
```

⟨*Map uniform random numbers to* $[-1, 1]^2$⟩ $\equiv$ **667**

```
float sx = 2 * u1 - 1;
float sy = 2 * u2 - 1;
```

⟨*Map square to* (r, θ)⟩ ≡ **667**
```
⟨Handle degeneracy at the origin⟩
if (sx >= -sy) {
    if (sx > sy) {
        ⟨Handle first region of disk 668⟩
    }
    else {
        ⟨Handle second region of disk⟩
    }
}
else {
    if (sx <= sy) {
        ⟨Handle third region of disk⟩
    }
    else {
        ⟨Handle fourth region of disk⟩
    }
}
theta *= M_PI / 4.f;
```

⟨*Handle first region of disk*⟩ ≡ **668**
```
r = sx;
if (sy > 0.0) theta = sy/r;
else          theta = 8.0f + sy/r;
```

The remaining cases are analogous and are omitted.

13.6.3 EXAMPLE: COSINE-WEIGHTED HEMISPHERE SAMPLING

As we will see later in the next chapter, it is often useful to sample from a distribution that has a shape similar to that of the integrand being estimated. For example, because the scattering equation weights the product of the BSDF and the incident radiance with a cosine term, it is useful to have a method that generates directions that are more likely to be close to the top of the hemisphere, where the cosine term has a large value, than the bottom, where the cosine term is small.

Mathematically, this means that we would like to sample directions ω from a PDF

$$p(\omega) \propto \cos\theta .$$

Normalizing as usual,

$$\int_{\mathcal{H}^2} c\; p(\omega)\, \mathrm{d}\omega = 1$$

$$\int_0^{2\pi} \int_0^{\frac{\pi}{2}} c \cos\theta \sin\theta \,\mathrm{d}\theta \,\mathrm{d}\phi = 1$$

$$c\, 2\pi \int_0^{\pi/2} \cos\theta \sin\theta \,\mathrm{d}\theta = 1$$

$$c = \frac{1}{\pi}$$

M_PI 1002

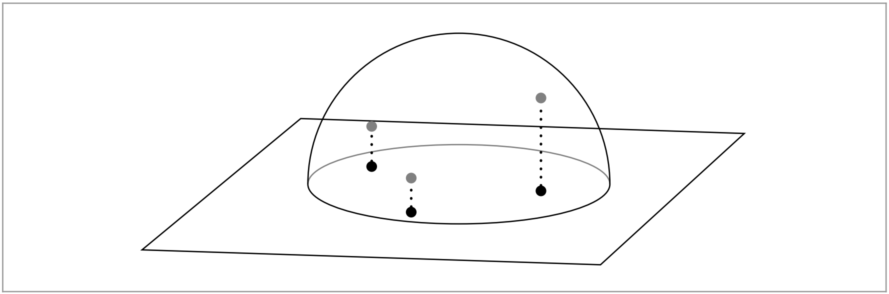


Figure 13.14: Malley's Method. To sample direction vectors from a cosine-weighted distribution, uniformly sample points on the unit disk and project them up to the unit sphere.

so

$$p(\theta, \phi) = \frac{1}{\pi} \cos\theta \sin\theta .$$

We could compute the marginal and conditional densities as before, but instead we can use a technique known as *Malley's method* to generate these cosine-weighted points. The idea behind Malley's method is that if we choose points uniformly from the unit disk and then generate directions by projecting the points on the disk up to the hemisphere above it, the resulting distribution of directions will have a cosine distribution (Figure 13.14).

Why does this work? Let (r, ϕ) be the polar coordinates of the point chosen on the disk (note that we're using ϕ instead of the usual θ here). From our calculations before, we know that the joint density $p(r, \phi) = r/\pi$ gives the density of a point sampled on the disk.

Now, we map this to the hemisphere. The vertical projection gives $\sin\theta = r$, which is easily seen from Figure 13.14. To complete the $(r, \phi) \to (\sin\theta, \phi)$ transformation, we need the determinant of the Jacobian

$$|J_T| = \begin{vmatrix} \cos\theta & 0 \\ 0 & 1 \end{vmatrix} = \cos\theta .$$

Therefore, $p(\theta, \phi) = |J_T| p(r, \phi) = \cos\theta r/\pi = (\cos\theta \sin\theta)/\pi$, which is exactly what we wanted! We have used the transformation method to prove that Malley's method generates directions with a cosine-weighted distribution. Note that this technique works regardless of the method used to sample points from the circle, so we can use Shirley's concentric mapping just as well as the simpler $(r, \theta) = (\sqrt{\xi_1}, 2\pi\xi_2)$ method.

⟨*Monte Carlo Utility Declarations*⟩ +≡

```
inline Vector CosineSampleHemisphere(float u1, float u2) {
    Vector ret;
    ConcentricSampleDisk(u1, u2, &ret.x, &ret.y);
    ret.z = sqrtf(max(0.f, 1.f - ret.x*ret.x - ret.y*ret.y));
    return ret;
}
```

ConcentricSampleDisk() 667
Vector 57

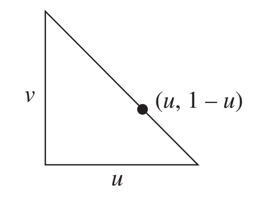


Figure 13.15: Sampling an Isosceles Right Triangle. Note that the equation of the hypotenuse is $v = 1 - u$.

Remember that all of the PDF evaluation routines in pbrt are defined with respect to solid angle, not spherical coordinates, so the PDF function returns a weight of $\cos\theta/\pi$.

⟨*Monte Carlo Utility Declarations*⟩ +≡
```
inline float CosineHemispherePdf(float costheta, float phi) {
    return costheta * INV_PI;
}
```

13.6.4 EXAMPLE: SAMPLING A TRIANGLE

Although uniformly sampling a triangle may seem like a simple task, it turns out to be more complex than the ones we've seen so far.[8] To simplify the problem, we will assume we are sampling an isosceles right triangle of area $\frac{1}{2}$. The output of the sampling routine that we will derive will be barycentric coordinates, however, so the technique will actually work for any triangle despite this simplification. Figure 13.15 shows the shape to be sampled.

We will denote the two barycentric coordinates here by (u, v). Since we are sampling with respect to area, we know that the PDF $p(u, v)$ must be a constant equal to the reciprocal of the shape's area, $\frac{1}{2}$, so $p(u, v) = 2$.

First, we find the marginal density $p(u)$:

$$p(u) = \int_0^{1-u} p(u, v)\, dv = 2 \int_0^{1-u} dv = 2(1-u),$$

and the conditional density $p(v|u)$:

$$p(v|u) = \frac{p(u, v)}{p(u)} = \frac{2}{2(1-u)} = \frac{1}{1-u}.$$

The CDFs are, as always, found by integration:

8 It is possible to generate the right distribution in a triangle by sampling the enclosing parallelogram and reflecting samples on the wrong side of the diagonal back into the triangle. Although this technique is simpler than the one presented here, it is undesirable since it effectively folds the 2D uniform random samples back on top of each other—two samples that are very far away (e.g., $(.01, .01)$ and $(.99, .99)$) can map to the same point on the triangle. This thwarts variance reduction techniques like stratified sampling that generate sets of well-distributed (ξ_1, ξ_2) samples and expect that they will map to well-distributed points on the object being sampled; see Section 14.2.1 for further discussion.

INV_PI 1002

$$P(u) = \int_0^u p(u')\, \mathrm{d}u' = 2u - u^2$$

$$P(v) = \int_0^v p(v'|u)\, \mathrm{d}v' = \frac{v}{1-u}.$$

Inverting these functions and assigning them to uniform random variables gives the final sampling strategy:

$$u = 1 - \sqrt{\xi_1}$$
$$v = \xi_2\sqrt{\xi_1}.$$

Notice that the two variables in this case are *not* independent!

⟨*Monte Carlo Function Definitions*⟩ +≡

```
void UniformSampleTriangle(float u1, float u2, float *u, float *v) {
    float su1 = sqrtf(u1);
    *u = 1.f - su1;
    *v = u2 * su1;
}
```

We won't provide a PDF evaluation function for this sampling strategy since the appropriate value depends on the triangle's area.

13.6.5 EXAMPLE: PIECEWISE-CONSTANT 2D DISTRIBUTIONS

Our final example will show how to sample from discrete 2D distributions. We will consider the case of a 2D function defined over $(u, v) \in [0, 1]^2$ by a 2D array of $n_u \times n_v$ sample values. This case is particularly useful for generating samples from distributions defined by texture maps and environment maps.

Consider a 2D function $f(u, v)$ defined by a set of $n_u \times n_v$ values $f[u_i, v_j]$ where $u_i \in [0, 1, \ldots, n_u - 1]$, $v_j \in [0, 1, \ldots, n_v - 1]$, and $f[u_i, v_j]$ give the constant value of f over the range $[i/n_u, (i+1)/n_u) \times [j/n_v, (j+1)/n_v)$. Given continuous values (u, v), we will use $(\tilde{u}, \tilde{v})$ to denote the corresponding discrete (u_i, v_j) indices, with $\tilde{u} = \lfloor n_u u \rfloor$ and $\tilde{v} = \lfloor n_v v \rfloor$ so that $f(u, v) = f[\tilde{u}, \tilde{v}]$.

Integrals of f are simple sums of $f[u_i, v_j]$ values, so that, for example, the integral of f over the domain is

$$I_f = \iint f(u, v)\, \mathrm{d}u\, \mathrm{d}v = \frac{1}{n_u n_v} \sum_{i=0}^{n_u-1} \sum_{j=0}^{n_v-1} f[u_i, v_j].$$

Using the definition of the PDF and the integral of f, we can find f's PDF,

$$p(u, v) = \frac{f(u, v)}{\iint f(u, v)\, \mathrm{d}u\, \mathrm{d}v} = \frac{f[\tilde{u}, \tilde{v}]}{1/(n_u n_v) \sum_i \sum_j f[u_i, v_j]}.$$

Recalling Equation (13.12), the marginal density $p(v)$ can be computed as a sum of $f[u_i, v_j]$ values

$$p(v) = \int p(u, v)\, \mathrm{d}u = \frac{(1/n_u) \sum_i f[u_i, \tilde{v}]}{I_f}. \tag{13.14}$$

Because this function only depends on $\tilde{v}$, it is thus itself a piecewise constant 1D function, $p[\tilde{v}]$, defined by n_v values. The 1D sampling machinery from Section 13.3.1 can be applied to sampling from its distribution.

Given a v sample, the conditional density $p(u|v)$ is then

$$p(u|v) = \frac{p(u, v)}{p(v)} = \frac{f[\tilde{u}, \tilde{v}]/I_f}{p[\tilde{v}]}. \qquad (13.15)$$

Note that, given a particular value of $\tilde{v}$, $p[\tilde{u}|\tilde{v}]$ is a piecewise-constant 1D function of $\tilde{u}$, that can be sampled with the usual one-dimensional approach. There are n_v such distinct 1D conditional densities, one for each possible value of $\tilde{v}$.

Putting this all together, the `Distribution2D` structure provides functionality similar to `Distribution1D`, except that it generates samples from piecewise-constant 2D distributions.

```
⟨Monte Carlo Utility Declarations⟩ +≡
    struct Distribution2D {
        ⟨Distribution2D Public Methods 673⟩
    private:
        ⟨Distribution2D Private Data 672⟩
    };
```

The constructor has two tasks. First, it computes a 1D conditional sampling density $p[\tilde{u}|\tilde{v}]$ for each of the n_v individual $\tilde{v}$ values using Equation (13.15). It then computes the marginal sampling density $p[\tilde{v}]$ with Equation (13.14).

```
⟨Monte Carlo Function Definitions⟩ +≡
    Distribution2D::Distribution2D(const float *func, int nu, int nv) {
        for (int v = 0; v < nv; ++v) {
            ⟨Compute conditional sampling distribution for ṽ 672⟩
        }
        ⟨Compute marginal sampling distribution p[ṽ] 673⟩
    }
```

First the $p[\tilde{u}|\tilde{v}]$ distributions are computed. `Distribution1D` can do this directly by being given a pointer to each of the n_v rows of n_u function values, since they're laid out linearly in memory. The I_f and $p[\tilde{v}]$ terms from Equation (13.15) don't need to be included in the values passed to `Distribution1D`, since they have the same value for all of the n_u values and are thus just a constant scale that doesn't affect the normalized distribution that `Distribution1D` computes.

```
⟨Compute conditional sampling distribution for ṽ⟩ ≡                    672
    pConditionalV.push_back(new Distribution1D(&func[v*nu], nu));
```

```
⟨Distribution2D Private Data⟩ ≡                                         672
    vector<Distribution1D *> pConditionalV;
```

Distribution1D 648
Distribution2D 672
Distribution2D::pConditionalV 672

Given the conditional densities for each $\tilde{v}$ value, we can find the 1D marginal density for sampling each $\tilde{v}$ value, $p[\tilde{v}]$. The `Distribution1D` class stores the integral of the piecewise-constant function it represents in its `funcInt` member variable, so it's just necessary to copy these values to the `marginalFunc` buffer so they're stored linearly in memory for the `Distribution1D` constructor.

⟨*Compute marginal sampling distribution* $p[\tilde{v}]$⟩ ≡ 672

```
vector<float> marginalFunc;
for (int v = 0; v < nv; ++v)
    marginalFunc.push_back(pConditionalV[v]->funcInt);
pMarginal = new Distribution1D(&marginalFunc[0], nv);
```

⟨*Distribution2D Private Data*⟩ +≡ 672

```
Distribution1D *pMarginal;
```

As described previously, in order to sample from the 2D distribution, first a sample is drawn from the $p[\tilde{v}]$ marginal distribution in order to find the v coordinate of the sample. This floating-point value can be converted to the integer $\tilde{v}$ value in order to determine which of the precomputed condititional distributions should be used for sampling the u value. Figure 13.16 illustrates this idea using a low-resolution image as an example.

⟨*Distribution2D Public Methods*⟩ ≡ 672

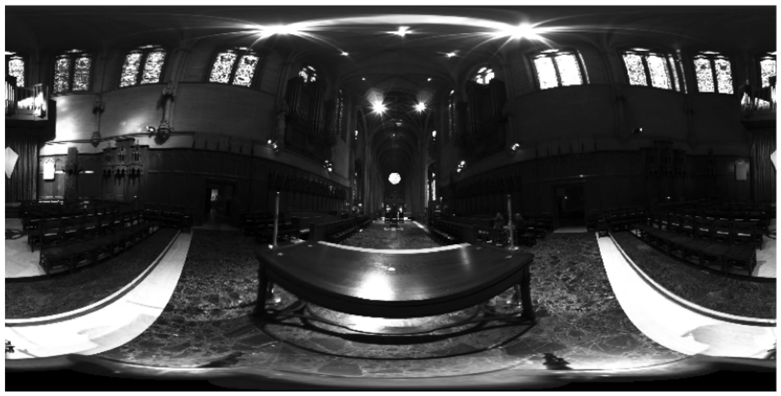
```
void SampleContinuous(float u0, float u1, float uv[2],
                      float *pdf) const {
    float pdfs[2];
    uv[1] = pMarginal->SampleContinuous(u1, &pdfs[1]);
    int v = Clamp(Float2Int(uv[1] * pMarginal->count), 0,
                  pMarginal->count-1);
    uv[0] = pConditionalV[v]->SampleContinuous(u0, &pdfs[0]);
    *pdf = pdfs[0] * pdfs[1];
}
```

The value of the PDF for a given sample value is computed as the product of the conditional and marginal PDFs for sampling it from the distribution.

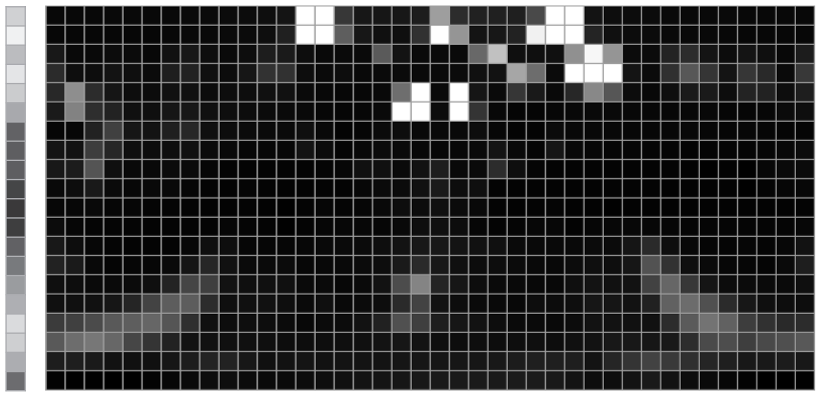
⟨*Distribution2D Public Methods*⟩ +≡ 672

```
float Pdf(float u, float v) const {
    int iu = Clamp(Float2Int(u * pConditionalV[0]->count), 0,
                   pConditionalV[0]->count-1);
    int iv = Clamp(Float2Int(v * pMarginal->count), 0,
                   pMarginal->count-1);
    if (pConditionalV[iv]->funcInt * pMarginal->funcInt == 0.f) return 0.f;
    return (pConditionalV[iv]->func[iu] * pMarginal->func[iv]) /
           (pConditionalV[iv]->funcInt * pMarginal->funcInt);
}
```

Clamp() 1000
Distribution1D 648
Distribution1D::count 648
Distribution1D::func 648
Distribution1D::funcInt 648
Distribution1D::SampleContinuous() 649
Distribution2D::pConditionalV 672
Distribution2D::pMarginal 673
Float2Int() 1002



(a)



(b) (c)

Figure 13.16: The Piecewise-Constant Sampling Distribution for a High-Dynamic-Range Environment Map. (a) The original environment map, (b) a low-resolution version of the marginal density function $p[\hat{v}]$, (c) the conditional distributions for rows of the image. First the marginal 1D distribution (b) is used to select a v value, giving a row of the image to sample. Rows with bright pixels are more likely to be sampled. Then, given a row, a value u is sampled from that row's 1D distribution. *(Grace Cathedral environment map courtesy of Paul Debevec.)*

FURTHER READING

Many books have been written on Monte Carlo integration. Hammersley and Handscomb (1964), Spanier and Gelbard (1969), and Kalos and Whitlock (1986) are classic references. More recent books on the topic include those by Fishman (1996) and Liu (2001). Chib and Greenberg (1995) have written an approachable but rigorous introduction to the Metropolis algorithm. The Monte Carlo and Quasi Monte Carlo Web site is a useful gateway to recent work in the field *(www.mcqmc.org)*.

Good general references about Monte Carlo and its application to computer graphics are the theses by Lafortune (1996) and Veach (1997). This topic is also covered well by Dutré, Bekaert, and Bala (2006). Dutré's *Global Illumination Compendium* (2003) also has much useful information related to this topic. The course notes from the Monte Carlo raytracing course at SIGGRAPH also have a wealth of practical information (Jensen et al. 2001a, 2003). The square to disk mapping was described by Shirley and Chiu (1997). Clarberg (2008) described a highly efficient implementation of a mapping from the square to the sphere that shares many of the important properties identified by Shirley and Chiu.

Steigleder and McCool (2003) described an alternative to the multidimensional sampling approach from Section 13.6.5: they linearized two and higher dimensional domains into 1D using a Hilbert curve and then sampled using 1D samples over the 1D domain. This leads to a simpler implementation that still maintains desirable stratification properties of the sampling distribution, thanks to the spatial coherence preserving properties of the Hilbert curve.

Lawrence et al. (2005) described an adaptive representation for CDFs, where the CDF is approximated with a piecewise linear function with fewer, but irregularly spaced, vertices compared to the complete CDF. This approach can substantially reduce storage requirements and improve lookup efficiency, taking advantage of the fact that large ranges of the CDF may be efficiently approximated with a single linear function.

Cline et al. (2009b) observed that the time spent in binary searches for sampling from precomputed distributions (like `Distribution1D` does) can take a substantial amount of execution time. (Indeed, `pbrt` spends approximately 18% of the time when rendering the TT car scene lit by an `InfiniteAreaLight` in the `Distribution1D::SampleContinuous()` method, which is used by `InfiniteAreaLight::Sample_L()`.) They presented two improved methods for doing this sort of search: the first stores a lookup table with n integer values, indexed by $\lfloor n\xi \rfloor$, which gives the first entry in the CDF array that is less than or equal to ξ. Starting a linear search from this offset in the CDF array can be much more efficient than a complete binary search over the entire array. They also presented a method based on approximating the inverse CDF as a piecewise linear function of ξ and thus enabling constant-time lookups at a cost of some accuracy (and thus some additional variance).

EXERCISES

❷ 13.1 Write a program that compares Monte Carlo and one or more alternative numerical integration techniques. Structure this program so that it is easy to replace the particular function being integrated. Verify that the different techniques compute the same result (given a sufficient number of samples for each of them). Modify your program so that it draws samples from distributions other than the uniform distribution for the Monte Carlo estimate and verify that it still computes the correct result when the correct estimator, Equation (13.3), is used. (Make sure that any alternative distributions you use have nonzero probability of choosing any value of x where $f(x) > 0$.)

Distribution1D 648
Distribution1D::SampleContinuous() 649
InfiniteAreaLight 629
InfiniteAreaLight::Sample_L() 727

❶ 13.2 Write a program that computes Monte Carlo estimates of the integral of a given function. Compute an estimate of the variance of the estimates by taking a series of trials and using Equation (13.2) to compute variance. Demonstrate numerically that variance decreases at a rate of $O(\sqrt{n})$.

❶ 13.3 The depth-of-field code in Section 6.2.3 uses the `ConcentricSampleDisk()` function to generate samples on the circular lens, since this function gives less distortion than `UniformSampleDisk()`. Try replacing it with `UniformSampleDisk()` and measure the difference in image quality. For example, you might want to compare the error in images from using each approach and a relatively low number of samples to a highly sampled reference image.

Does `ConcentricSampleDisk()` in fact give less error in practice? Does it make a difference if a relatively simple scene is being rendered versus a very complex scene?

❷ 13.4 Modify the `Distribution1D` implementation to use the adaptive CDF representation described by Lawrence et al. (2005) and experiment with how much more compact the CDF representation can be made without causing image artifacts. (Good test scenes include those that use `InfiniteAreaLights`, which use the `Distribution2D` and, thus, `Distribution1D` for sampling.) Can you measure an improvement in rendering speed due to more efficient searches through the approximated CDF?

ConcentricSampleDisk() 667
Distribution1D 648
Distribution2D 672
InfiniteAreaLight 629
UniformSampleDisk() 666