# 2 Markov chain Monte Carlo

***This chapter covers***

- Introducing the Markov chain Monte Carlo
- Estimating pi via Monte Carlo integration
- Binomial tree model Monte Carlo simulation
- Self-avoiding random walk
- Gibbs sampling algorithm
- Metropolis-Hastings algorithm
- Importance sampling

In the previous chapter, we reviewed different types of ML algorithms and software implementation. Now, we will focus on a popular class of ML algorithms known as Markov chain Monte Carlo. Any probabilistic model that explains a part of reality exists in a high-dimensional parameter space because it is described by high dimensional model parameters. *Markov chain Monte Carlo* (MCMC) is a methodology of sampling from high-dimensional parameter spaces to approximate the posterior distribution $p(\theta|x)$. Originally developed by physicists, this method became popular in the Bayesian statistics community because it allows one to estimate high dimensional posterior distributions using sampling. The basic idea behind MCMC is to construct a Markov chain whose stationary distribution is equal to the target posterior $p(\theta|x)$.

In other words, if we perform a random walk across the parameter space, the fraction of time we spend in a particular state $\theta$ is proportional to $p(\theta|x)$.

We begin by introducing MCMC in the following section. We'll proceed with three warm-up Monte Carlo examples (estimating pi, binomial tree model, and self-avoiding random walk) before looking at three popular sampling algorithms (Gibbs sampling, Metropolis-Hastings, and importance sampling). The warm-up algorithms in this chapter are selected to give the reader an appropriate introduction to MCMC, and the sampling algorithms are selected for their wide application in sampling from probabilistic graphical models (PGMs).

## 2.1 Introduction to Markov chain Monte Carlo

Let's start by understanding high dimensional parameter spaces based on a simple example of classifying irises. An *iris* is a species of flowers consisting of three types: setosa, versicolor, and virginica. The flowers are characterized by their petal and sepal length and width, which can be used as features to determine the iris type. Figure 2.1 shows the iris pairplot.

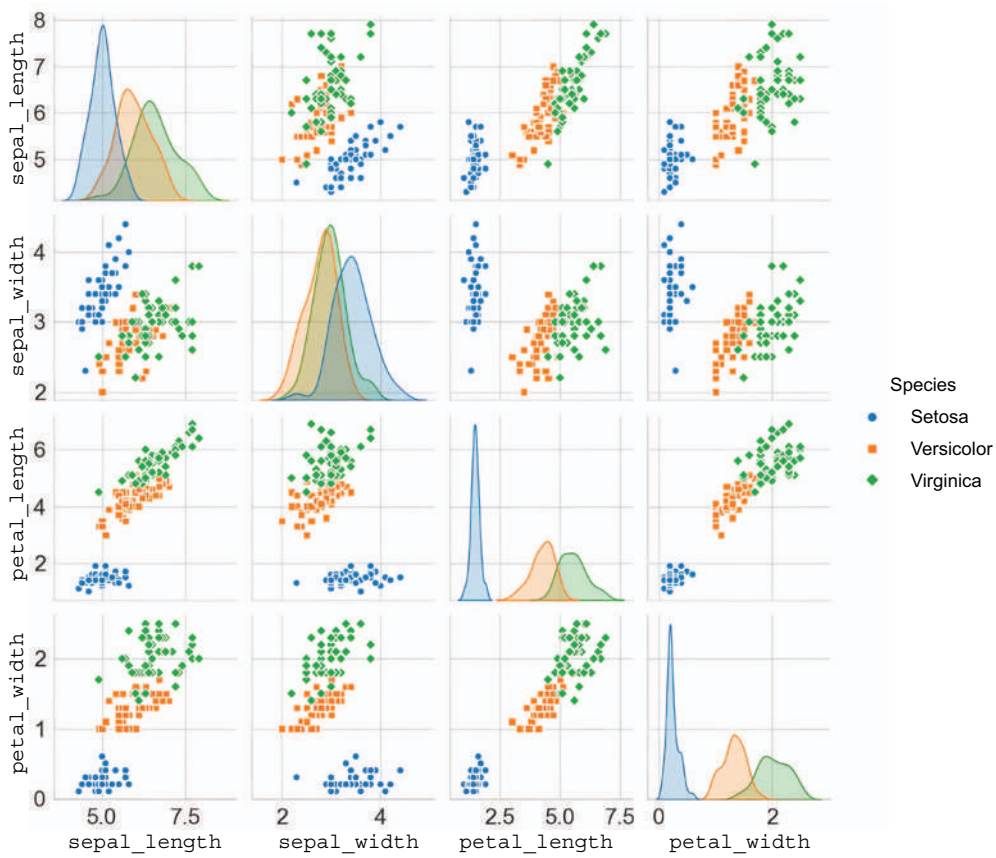


**Figure 2.1 Iris pairplot: pairwise scatterplots color coded by iris species**

A pairplot is a matrix of plots where off-diagonal entries contain a scatter plot of every feature (e.g., petal length) against every other feature, while the main diagonal entries contain plots of every feature against itself color-coded by the three iris species. The pairplot captures pairwise relationships in the iris dataset. Let's focus on the main diagonal entries and try to model the data we see. Since we have three types of Iris flowers, we can model the data as a mixture of three Gaussian distributions. A Gaussian mixture is a weighted sum of Gaussian probability distributions. Equation 2.1 captures this in mathematical terms.

Sum of K Gaussians

$$p(x|\theta) = \sum_{k=1}^{K} N\left(x; \mu_k, \sigma_k^2\right) \pi_k \tag{2.1}$$

Gaussian mixture

Gaussian RV

Mixture proportions

A mixture of Gaussians consists of K Gaussian RVs scaled by mixture proportions $\pi k$ that are positive and add up to 1: $\sum \pi_k = 1$, $\pi_k > 0$. This is a high dimensional parameter problem because to fit the Gaussian mixture model, we need to find the values for the means $\mu_k$, covariances $\sigma_k^2$, and mixture proportions $\pi_k$. In case of the iris dataset, we have K = 3, which means the number of parameters for the Gaussian mixture is equal to 9 – 1 = 8 (i.e., we have 9 parameters in $\theta = \{\mu_k, \sigma_k^2, \pi_k\}_{k=1}^{3}$, and we subract 1 because of the sum-to-one constraint for $\pi k$. In a later chapter, we'll look at how to find the Gaussian mixture parameters via the expectation-maximization (EM) algorithm. Now that we have an idea about parameter spaces, let's review our understanding of posterior distribution, using coin flips as an example.

### 2.1.1 Posterior distribution of coin flips

With MCMC algorithms, we attempt to approximate the posterior distribution through samples. In fact, most of the Bayesian inference is designed to efficiently approximate the posterior distribution. Let's understand what a posterior distribution is in a little more detail. Posterior arises when we have a model with parameters $\theta$ we are trying to fit to observed data x. Namely, it's the probability of parameters given the data $p(\theta|x)$. Consider an example of a sequence of coin flips where every coin is heads with probability $\theta$ and tails with probability $1-\theta$, as shown in figure 2.2.

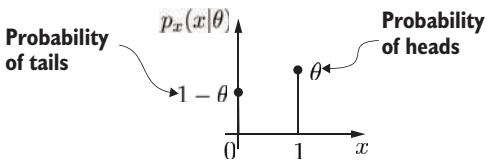


Figure 2.2 Bernoulli random variable

Figure 2.2 shows the probability mass function (PMF) of a Bernoulli RV modeling the coin flip. If $\theta = ½$, we have a fair coin with an equal chance of heads (1) or tails (0); otherwise, we say that the coin is biased, with a bias equal to $\theta$. We can write down the PMF of a Bernoulli RV as follows.

$$p_x(x|\theta) = \theta^x(1-\theta)^{1-x}, x \in \{0, 1\} \tag{2.2}$$

The mean of the Bernoulli RV is equal to $\mathrm{E}[X] = \sum_x x p_x(x|\theta) = 0(1-\theta) + 1(\theta) = \theta$, while the variance is equal to $\mathrm{VAR}(x) = \mathrm{E}[x^2] - \mathrm{E}[x]^2 = \theta - \theta^2 = \theta(1-\theta)$. Let's go back to our coin-tossing example. Let $D = \{x_1,\ldots,x_n\}$ be a sequence of independent and identically distributed (iid) coin flips. Then, we can compute the likelihood as follows.

$$\begin{aligned} p_x(D|\theta) &= p_x(x_1, \ldots, x_n|\theta) \\ &= \prod_{i=1}^{n} p(x_i|\theta) = \prod_{i=1}^{n} \theta^{x_i}(1-\theta)^{1-x_i} \\ &= \theta^{N_1}(1-\theta)^{N_0} \end{aligned} \tag{2.3}$$

Here, we assume $N_1 = \sum_{i=1}^{n} x_i$, or the total number of counts for which the coin landed heads, and $N_0 = n - N_1$, or the total number of counts for which the coin landed tails.

Equations 2.2 and 2.3 have the form $p(x|\theta)$, which is, by definition, the likelihood of data $x$, given the parameter $\theta$. What if we have some prior information about the parameter $\theta$; in other words, what if we know something about $p(\theta)$, such as the number of heads and tails we obtained in another experiment with the same coins? We can capture this prior information as follows.

$$\mathrm{Beta}(\theta|a, b) \propto \theta^{a-1}(1-\theta)^{b-1} \tag{2.4}$$

Here, a is the number of heads and b is the number of tails in our previous experiment. When computing the posterior distribution, we are interested in computing $p(\theta|x)$. We can use the Bayes rule from chapter 1 to express the posterior distribution as proportional to the product of the likelihood and prior.

$$\begin{aligned} p(\theta|D) &\propto p(D|\theta)p(\theta) \\ &= \theta^{N_1}(1-\theta)^{N_0}\theta^{a-1}(1-\theta)^{b-1} = \theta^{N_1+a-1}(1-\theta)^{N_0+b-1} \\ &\propto \mathrm{Beta}(\theta|N_1 + a, N_0 + b) \end{aligned} \tag{2.5}$$

Equation 2.5 computes the posterior distribution $p(\theta|D)$, which tells us the probability of heads for a sequence of coin flips. We can see that the posterior is distributed as a Beta random variable. Remember that our prior was also a Beta random variable; we

say that the prior is "conjugate to the posterior." In other words, the posterior can be computed in closed form by updating the prior counts with observed data.

### 2.1.2 Markov chain for page rank

Before we dive into MCMC algorithms, we need to introduce the concept of a Markov chain. A *Markov chain* is a sequence of possible events, in which the probability of the current event depends only on the previous event. A first-order Markov chain can be written as shown in equation 2.6. Let $x_1,\ldots,x_t$ be the samples from our posterior distribution; then, we can write the joint as follows.

$$\begin{aligned} p(x_1, \ldots, x_t) &= p(x_1)p(x_2|x_1)p(x_3|x_2)\cdots p(x_t|x_{t-1}) \\ &= p(x_1)\prod_{i=2}^{t} p(x_i|x_{i-1}) \end{aligned} \tag{2.6}$$

Notice how the joint factors as a product of conditional distributions, where the current state is conditioned only on the previous state. A Markov chain is characterized by the initial distribution over the states $p(x_1 = i)$ and a state transition matrix $A_{ij} = p(x_t = j\,|x_{t-1} = i)$ from state i to state j.

Let's motivate Markov chains via a Google page rank example. Google uses a page rank algorithm to rank billions of web pages. We can formulate a collection of $n$ web pages as a graph G = (V, E). Let every node $v \in$ V represent a web page and every edge $e \in$ E represent a link from one page to another. Knowing how the pages are linked allows us to construct a giant, sparse transition matrix $A$, where $A_{ij}$ is the probability of following a link from page $i$ to page $j$, as shown in figure 2.3.

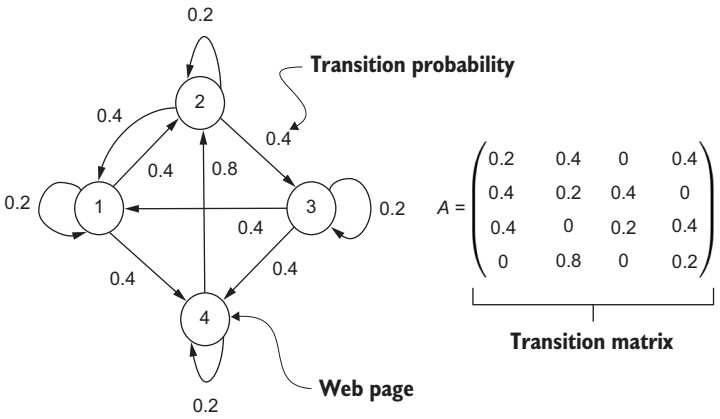


**Figure 2.3 A graph of web pages with transition probabilities**

Note that every row in the matrix A sums to 1 (i.e., $\sum_j A_{ij} = 1$). A matrix with this property is known as a *stochastic matrix*. Note that $A_{ij}$ corresponds to transition probability from state $i$ to state $j$. As we will see in a later chapter, page rank is a stationary distribution over the states of the Markov chain. To make it scale to billions of web pages, we'll

derive from scratch and implement a power iteration algorithm. But for now, in the next few sections, we will look at different MCMC sampling algorithms: Gibbs sampling, Metropolis-Hastings, and importance sampling. Let's start with a couple of warm-up examples as our first exposure to Monte Carlo algorithms.

## 2.2 Estimating pi

The first example we will look at is estimating the value of $\pi$ via Monte Carlo integration. *Monte Carlo* (MC) *integration* has an advantage over numerical integration (which evaluates a function at a fixed grid of points) in that the function is only evaluated in places where there is nonnegligible probability. Thus, MC integration scales better to high-dimensional problems.

Let's look at the expected value of some function **f:** $\mathbb{R} \rightarrow \mathbb{R}$ of a random variable $Z = f(Y)$. We can approximate it by drawing samples y from the distribution $p(y)$ as follows.

$$E[f(Y)] = \int f(y)p(y)dy \approx \frac{1}{S}\sum_{s=1}^{S} f(y_s), \text{ where } y_s \sim p(y) \tag{2.7}$$

In other words, we are approximating the expected value $E[f(y)]$ with an average of $f(y_s)$, where $y_s$ are samples drawn from the distribution $p(y)$. Let's now use the same idea but apply it to evaluating an integral $I = \int f(x)\,dx$. We can approximate the integral as follows.

$$\begin{aligned} I &= \int_a^b f(x)dx = \int_a^b w(x)p(x)dx = E_p[w(x)] \\ &= \frac{1}{n}\sum_{i=1}^{N} w(x_i), \text{ where } x_i \sim p(x) \end{aligned} \tag{2.8}$$

Here, $p(x) \sim \text{Unif}(a, b) = 1/(b-a)$ is the PDF of a uniform random variable over the interval $(a, b)$ and $w(x) = f(x)(b-a)$ is our scaled function $f(x)$.

As we increase the number of samples $N$, our empirical estimate of the mean becomes more accurate. In fact, the standard error is equal to $\sigma/\text{sqrt}(N)$, where $\sigma$ is the empirical standard deviation.

$$\sigma^2 = \frac{1}{N-1}\sum_{i=1}^{N}(f(x_i) - I)^2 \tag{2.9}$$

Now that we have an idea of how Monte Carlo integration works, let's use it to estimate $\pi$. We know that the area of a circle with radius $r$ is $I = \pi r^2$. Alternatively, the area of the circle can be computed as an integral.

$$I = \int_{-r}^{r} \int_{-r}^{r} 1[x^2 + y^2 \leq r^2] dx\, dy = E_{x,y}[w(x,y)] \tag{2.10}$$

Here, $1[x^2 + y^2 \leq r^2]$ is an indicator function equal to 1 when a point is inside the circle of radius $r$ and equal to 0 otherwise. Therefore, $\pi = I / r^2$. To compute $I$, note the following equation.

$$\begin{aligned} w(x,y) &= (b_x - a_x)(b_y - a_y)\, 1\left[x^2 + y^2 \leq r^2\right] \\ &= (2r)(2r) 1\left[x^2 + y^2 \leq r^2\right] \\ &= 4r^2 1\left[x^2 + y^2 \leq r^2\right] \end{aligned} \tag{2.11}$$

We can summarize the pi estimation algorithm in the pseudo-code shown in figure 2.4.

1: **function** pi_est($R$, $N$):
2: **for** $i = 1$ to $N$:
3: $X[i] \sim \text{Unif}(-R, R)$ ┐
4: $Y[i] \sim \text{Unif}(-R, R)$ ┘ → **Uniform RVs**
5: $IN[i] = X[i]^2 + Y[i]^2 \leq R^2$
6: $S[i] = (2R) \times (2R) \times IN[i]$
7: **end for**
8: $\hat{I} = \frac{1}{N} \sum_{i=1}^{N} S[i]$
9: $\hat{\pi} = \frac{\hat{I}}{R^2}$ ← **Pi estimate**
10: $\hat{\pi}_{se} = \frac{\sigma_S}{\sqrt{N}}$ ← **Pi standard deviation**
11: **return** $\hat{\pi} \pm \hat{\pi}_{se}$

**Figure 2.4 Pi estimator pseudo-code**

We generate $N$ samples from a uniform distribution with support from $-R$ to $R$ and compute a Boolean expression of whether our sample is inside or outside the circle. If the sample is inside, it factors into the integral computation. Once we have an estimate of the integral, we divide the result by $R^2$ to compute our estimate of pi. We are now ready to implement our pi estimator!

**Listing 2.1 Pi estimator**

```
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(42)

def pi_est(radius=1, num_iter=int(1e4)):

    X = np.random.uniform(-radius,+radius,num_iter)
    Y = np.random.uniform(-radius,+radius,num_iter)
```

**Fixes a seed for reproducible results**

**Experiments with a different number of samples N**

```
    R2 = X**2 + Y**2
    inside = R2 < radius**2
    outside = ~inside

    samples = (2*radius)*(2*radius)*inside

    I_hat = np.mean(samples)
    pi_hat = I_hat/radius ** 2
    pi_hat_se = np.std(samples)/np.sqrt(num_iter)
    print("pi est: {} +/- {:f}".format(pi_hat, pi_hat_se))

    plt.figure()
    plt.scatter(X[inside],Y[inside], c='b', alpha=0.5)
    plt.scatter(X[outside],Y[outside], c='r', alpha=0.5)
    plt.show()

if __name__ == "__main__":

    pi_est()
```

Pi estimate

Pi standard deviation

If we execute the code, we get $\pi = 3.1348 \pm 0.0164$, which is a relatively good result (within the error bounds). Try experimenting with different numbers of samples $N$ to see how fast we converge to $\pi$ as a function of $N$. Figure 2.5 shows the accepted Monte Carlo samples in blue, corresponding to samples inside the circle, and the rejected samples in red, corresponding to samples outside the circle.

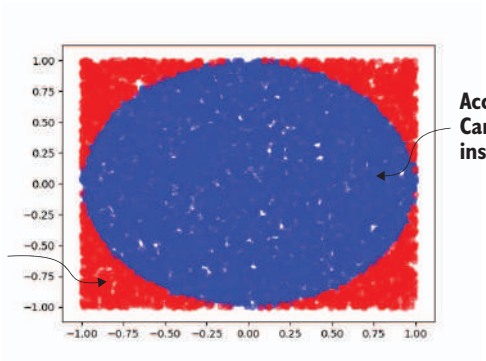


**Figure 2.5 Monte Carlo samples used to estimate pi**

In the preceding example, we assumed that samples came from a uniform distribution $X, Y \sim \text{Unif}(-R, R)$. In the following section, we will look at simulating a stock price over different time horizons using the binomial tree model.

## 2.3 Binomial tree model

Now, let's examine how Monte Carlo sampling can be used in finance. In a binomial tree model of a stock price, we assume that at each time step, the stock could be in either up or down states with unequal payoffs characteristic for a risky asset. Assuming the initial stock price at time $t = 0$ is \$1, at the next time step $t = 1$, the price is $u$ in the up state and $d$ in the down state, with up-state transition probability $p$. A binomial tree

model for the first two timesteps is shown in figure 2.6. Note that the price in the next up (down) state is $u$ ($d$) times the price of the previous state.

We can use Monte Carlo to generate uncertainty estimates of the terminal stock price after some time horizon $T$. When using a binomial model to describe the price process of the stock, we can use the following calibration (the derivation of which is outside the scope of this book).

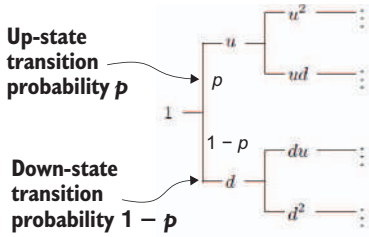


Figure 2.6 Binomial tree model

$$u = \exp\left(\sigma\sqrt{\frac{T}{n}}\right) = \frac{1}{d}$$
$$p = \frac{1}{2} + \frac{1}{2}\left(\frac{\mu}{\sigma}\right)\sqrt{\frac{T}{n}} \tag{2.12}$$

Here, $T$ is the length of the prediction horizon in years and n is the number of time steps. We assume 1 year equals 252 trading days, 1 month equals 21 days, 1 week equals 5 days, and 1 day equals 8 hours. Let's simulate the stock price using the binomial model with a daily time step for two different time horizons: 1 month from today and 1 year from today. We can summarize the algorithm in the pseudo-code shown in figure 2.7.

1: **function** binomial_tree($\mu$, $\sigma$, $S_0$, $N$, $T$, step):
2: $u = \exp\left(\sigma\sqrt{\frac{T}{n}}\right)$ ← **Up price**
3: $d = \frac{1}{u}$ ← **Down price**
4: $p = \frac{1}{2} + \frac{1}{2}\left(\frac{\mu}{\sigma}\right)\sqrt{\frac{T}{n}}$ ← **Up-state transition probability**
5: up_times = Binomial $\left(\frac{T}{\text{step}}, p, N\right)$
6: down_times = $\frac{T}{\text{step}}$ − up_times
7: $S_T = S_0 \times u^{up\_times} \times d^{down\_times}$
8: **return** $S_T$

Figure 2.7 Binomial tree pseudo-code

We begin by initializing up price $u$ and down price $d$, along with up-state transition probability $p$. Next, we simulate all the up-state transitions by sampling from a Binomial random variable with the number of trials equal to $T$/step, the success probability $p$, and the number of Monte Carlo simulations equal to $N$. The down-state transitions are computed by complimenting up-state transitions. Finally, we compute the asset price by multiplying the initial price $S_0$ with the price of all the up-state transitions and all the down-state transitions.

**Listing 2.2 Binomial tree stock price simulation**

```
import numpy as np

import seaborn as sns
import matplotlib.pyplot as plt

np.random.seed(42)

def binomial_tree(mu, sigma, S0, N, T, step):

    #compute state price and probability
    u = np.exp(sigma * np.sqrt(step))
    d = 1.0/u
    p = 0.5+0.5*(mu/sigma)*np.sqrt(step)

    #binomial tree simulation
    up_times = np.zeros((N, len(T)))
    down_times = np.zeros((N, len(T)))
    for idx in range(len(T)):
        up_times[:,idx] = np.random.binomial(T[idx]/step, p, N)
        down_times[:,idx] = T[idx]/step - up_times[:,idx]

    #compute terminal price
    ST = S0 * u**up_times * d**down_times

    #generate plots
    plt.figure()
    plt.plot(ST[:,0], color='b', alpha=0.5, label='1 month horizon')
    plt.plot(ST[:,1], color='r', alpha=0.5, label='1 year horizon')
    plt.xlabel('time step, day')
    plt.ylabel('price')
    plt.title('Binomial-Tree Stock Simulation')
    plt.legend()
    plt.show()

    plt.figure()
    plt.hist(ST[:,0], color='b', alpha=0.5, label='1 month horizon')
    plt.hist(ST[:,1], color='r', alpha=0.5, label='1 year horizon')
    plt.xlabel('price')
    plt.ylabel('count')
    plt.title('Binomial-Tree Stock Simulation')
    plt.legend()
    plt.show()

if __name__ == "__main__":

    #model parameters
    mu = 0.1
    sigma = 0.15
    S0 = 1
    N = 10000
    T = [21.0/252, 1.0]
    step = 1.0/252
    binomial_tree(mu, sigma, S0, N, T, step)
```

Up-state price

Down-state price

Probability of up state

Mean

Volatility

Starting price

Number of simulations

Time horizon, in years

Time step, in years

Figure 2.8 shows that our yearly estimates have higher volatility compared to the monthly estimates. This makes sense, as we expect to encounter more uncertainty over long time horizons. In the next section, we will learn how to use Monte Carlo to simulate self-avoiding random walks.

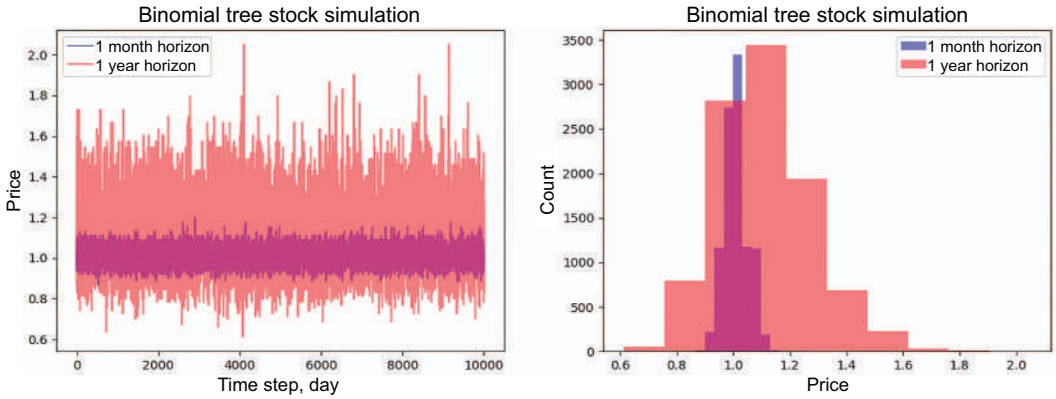


**Figure 2.8 Binomial tree stock simulation**

## 2.4 Self-avoiding random walk

Consider a random walk on a 2D grid. At each point on the grid, we take a random step with equal probability. This process forms a Markov chain $\{X_i\}_{i=1}^{n}$ on $\mathbb{Z}$ x $\mathbb{Z}$ with $X_0$=(0, 0) and transition probabilities given by equation 2.13.

$$P(X_i = (k,l)|X_{i-1} = (i,j)) = \begin{cases} \frac{1}{4}, & \text{if } |k-i| + |l-j| = 1 \\ 0, & \text{otherwise} \end{cases} \tag{2.13}$$

In other words, we have an equal probability of ¼ for transitioning from a point on the grid to any of the four—`up`, `down`, `left`, and `right`—neighbors. In addition, *self-avoiding* random walks are simply random walks that do not cross themselves. We can use Monte Carlo to simulate a self-avoiding random walk, as shown in the following pseudo-code (see figure 2.9).

We start by initializing the grid (lattice) to an all-zero matrix. We position the start of the random walk in the center of the grid, as represented by `xx` and `xx`. Notice that `num_step` is the number of steps in a random walk. We begin each iteration by computing the `up`, `down`, `left`, `right` values, which are equal to 1 if the grid is occupied and 0 otherwise. Therefore, we can compute available directions (neighbors) by computing 1 minus the direction. If the sum of neighbors is zero, there are no available directions (all the neighboring grid cells are occupied) and the random walk is self-intersecting, at which point we stop. Otherwise, we compute an importance weight as a product between the previous weight and the sum of neighbors. The importance weights are used to compute the weighted mean square distances of the random walk.

1: **function** rand_walk(num_step, num_iter, moves):
2: $X, Y$, lattice = 0, 0, 0
3: weight = 1
4: xx = num_step + 1 + $X$ ← **Middle of the *x*-axis**
5: yy = num_step + 1+ $Y$ ← **Middle of the *y*-axis**
6: lattice[xx, yy] = 1 ← **Init grid position**
7: **for** $i$ = 1 to num_step:
8: up = lattice[xx, yy+1]
9: down = lattice[xx, yy−1]
10: left = lattice[xx−1, yy]
11: right = lattice[xx+1, yy]
12: neighbors = [1, 1, 1, 1]− [up, down, left, right] ← **Available directions**
13: **if** sum(neighbors) == 0
14: break ← **Self-loop**
15: **end if**
16: weight = weight × sum(neighbors) ← **Computes importance weights**
17: direction ~ Cat $\left(\frac{\text{neighbors}}{\text{sum(neighbors)}}\right)$ ← **Samples a move direction**
18: $X$ = $X$+ moves[direction]
19: $Y$ = $Y$+ moves[direction]
20: // update grid coordinates
21: xx = num_step + 1 + $X$
22: yy = num_step + 1 + $Y$
23: lattice[xx, yy] = 1
24: **end for**
25: **return** lattice

**Figure 2.9 Random walk pseudo-code**

Next, we sample a move direction from the available directions, represented by a sample from the categorical random variable. Since the neighbors array can only take the values of 0 and 1, we are sampling uniformly from the available directions.

Next, we update the grid coordinates `xx` and `yy` and mark the occupancy by setting `lattice[xx, yy]=1`.While only a single iteration is shown in the pseudo-code, the following code listing wraps the pseudo-code in an additional for loop over the number of iterations `num_iter`. Each iteration is an attempt or a trial to produce a nonintersecting (i.e., self-avoiding random walk). The square distance of the random walk and the importance weights are recorded in each trial. We are now ready to look at the self-avoid random walk code.

**Listing 2.3 Self-avoiding random walk**

```
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

np.random.seed(42)

def rand_walk(num_step, num_iter, moves):
```

```
#random walk stats
square_dist = np.zeros(num_iter)
weights = np.zeros(num_iter)

for it in range(num_iter):

    trial = 0
    i = 1

    while i != num_step-1:

        #init
        X, Y = 0, 0
        weight = 1
        lattice = np.zeros((2*num_step+1, 2*num_step+1))
        lattice[num_step+1,num_step+1] = 1
        path = np.array([0, 0])
        xx = num_step + 1 + X
        yy = num_step + 1 + Y

        print("iter: %d, trial %d" %(it, trial))

        for i in range(num_step):

            up    = lattice[xx,yy+1]
            down  = lattice[xx,yy-1]
            left  = lattice[xx-1,yy]
            right = lattice[xx+1,yy]

            neighbors = np.array([1, 1, 1, 1]) -
            ➥ np.array([up, down, left, right])

            if (np.sum(neighbors) == 0):
                i = 1
                break
            #end if

            weight = weight * np.sum(neighbors)

            direction = np.where(np.random.rand() <
            ➥ np.cumsum(neighbors/float(sum(neighbors))))

            X = X + moves[direction[0][0],0]
            Y = Y + moves[direction[0][0],1]

            path_new = np.array([X,Y])
            path = np.vstack((path,path_new))

            #update grid coordinates
            xx = num_step + 1 + X
            yy = num_step + 1 + Y
            lattice[xx,yy] = 1
        #end for

        trial = trial + 1
```

Iterates until we have a noncrossing random walk

Computes available directions

Avoids self-loops

Computes importance weights

Samples a move direction

Stores a sampled path

```
    #end while

    square_dist[it] = X**2 + Y**2

    weights[it] = weight
#end for

mean_square_dist = np.mean(weights * square_dist)/np.mean(weights)
print("mean square dist: ", mean_square_dist)

#generate plots
plt.figure()
for i in range(num_step-1):
    plt.plot(path[i,0], path[i,1], path[i+1,0], path[i+1,1], 'ob')
plt.title('random walk with no overlaps')
plt.xlabel('X')
plt.ylabel('Y')
plt.show()

plt.figure()
sns.displot(square_dist)
plt.xlim(0,np.max(square_dist))
plt.title('square distance of the random walk')
plt.xlabel('square distance (X^2 + Y^2)')
plt.show()

if __name__ == "__main__":
    num_step = 150
    num_iter = 100
    moves = np.array([[0, 1],[0, -1],[-1, 0],[1, 0]])

    rand_walk(num_step, num_iter, moves)
```

Computes square extension

Number of iterations for averaging results

Number of steps in a random walk

2D moves

Figure 2.10 shows a self-avoiding random walk generated by the last iteration. There are $4^n$ possible random walks of length $n$ on the 2D lattice, and for large n, it is very unlikely a random walk will be self-avoiding. The figure also shows a histogram of the

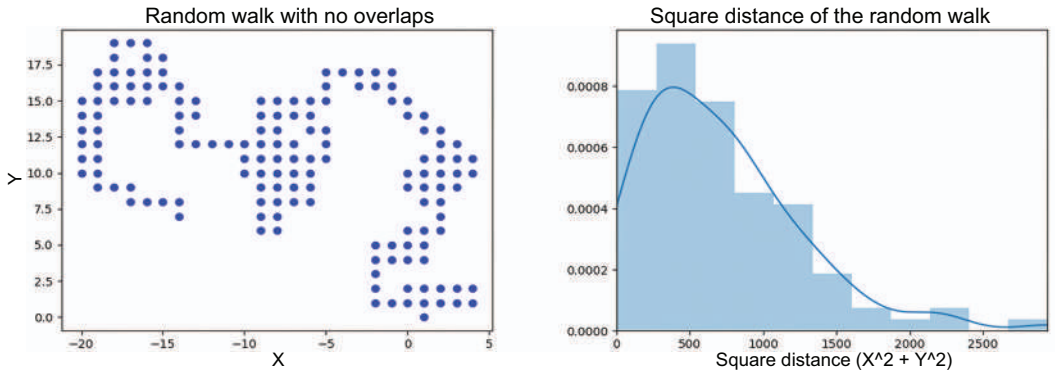


Figure 2.10 Self-avoiding random walk (left) and random walk square distance (right)

square distance computed for each random walk iteration. The histogram is positively skewed, showing a smaller probability of large-distance walks. In the next section, we will cover a popular sampling algorithm: Gibbs sampling.

## 2.5 Gibbs sampling

In this section, we introduce one of the fundamental MCMC algorithms, called *Gibbs sampling*. This form of sampling is based on the idea of sampling one variable at a time from a multidimensional distribution conditioned on the latest samples from all the other variables. For example, for a $d = 3$ dimensional distribution, given a starting sample $x^{\{k\}}$, we generate the next sample $x^{\{k+1\}}$ as follows.

$$\begin{aligned} x_1^{k+1} &\sim p\left(x_1 | x_2^k, x_3^k\right) \\ x_2^{k+1} &\sim p\left(x_2 | x_1^{k+1}, x_3^k\right) \\ x_3^{k+1} &\sim p\left(x_3 | x_1^{k+1}, x_2^{k+1}\right) \end{aligned} \tag{2.14}$$

The distributions in equation 2.14 are called *fully conditional distributions*. Also, notice that the naive Gibbs sampling algorithm is sequential (with the number of steps proportional to the dimensionality of the distribution) and it assumes we can easily sample from the fully conditional distributions. The Gibbs sampling algorithm is applicable to scenarios in which full conditional distributions in equation 2.14 are easy to compute.

One instance in which the fully conditional distributions are easy to compute is in the case of multivariate Gaussians with PDF, defined as follows.

$$\mathcal{N}(x; \mu, \Sigma) = \frac{1}{(2\pi)^{\frac{d}{2}} |\Sigma|^{\frac{1}{2}}} \exp\left[-\frac{1}{2}(x-\mu)^T \Sigma^{-1} (x-\mu)\right] \tag{2.15}$$

Let's partition the Gaussian vector into two sets $x\{1{:}D\} = (x_A, x_B)$ with the parameters shown in equation 2.16.

$$\begin{aligned} \mu &= \begin{pmatrix} \mu_A \\ \mu_B \end{pmatrix} \\ \Sigma &= \begin{pmatrix} \sum_{AA} & \sum_{AB} \\ \sum_{BA} & \sum_{BB} \end{pmatrix} \end{aligned} \tag{2.16}$$

Then, it can be shown, as discussed in section 2.3 of Christopher M. Bishop's *Pattern Recognition and Machine Learning* (2006), that the full conditionals are given by equation 2.17.

$$
\begin{aligned}
p(x_A|x_B) &= \mathcal{N}\left(x_A|\mu_{A|B}, \Sigma_{A|B}\right) \\
\mu_{A|B} &= \mu_A + \Sigma_{AB}\Sigma_{BB}^{-1}\left(x_B - \mu_B\right) \\
\Sigma_{A|B} &= \Sigma_{AA} - \Sigma_{AB}\Sigma_{BB}^{-1}\Sigma_{BA}
\end{aligned} \tag{2.17}
$$

Let's understand the Gibbs sampling algorithm by looking at the pseudo-code in figure 2.11.

```
1: class gibbs_gauss
2: function gauss_conditional(mu, Sigma, setA, x):
3: setU = set(range(len(mu))) ←Universal set
4: setB = setU\setA
5: x_B, μ_A, μ_B = x[setB], mu[setA], mu[setB]
6: μ_A|B = μ_A + Σ_AB Σ_BB^-1 (x_B − μ_B)
7: Σ_A|B = Σ_AA − Σ_AB Σ_BB^-1 Σ_BA
8: return μ_A|B, Σ_A|B
9: function sample(mu, Sigma, xinit, num_samples):
10: x = xinit
11: dim = len(mu)
12: for s = 1 to num_samples:
13:    for d = 1 to dim:
14:       μ_A|B, Σ_A|B = gauss_conditional(mu, Sigma, set(d), x)
15:       x[d] ~ N(x; μ_A|B, Σ_A|B) ←Gibbs samples
16:    end for
17:    samples[s,:] = x
18: end for
19: return samples
```

**Figure 2.11**
**Gibbs sampler pseudo-code**

Our `gibbs_gauss` class contains two `functions`: `gauss_conditional` and `sample`. The `gauss_conditional` function computes the conditional Gaussian distribution $p(x_A|x_B)$ for any sets of variables `setA` and setB. `setA` is an input to the function, while `setB` is computed as a set difference between the universal set of dimension $D$ and `setA`. Recall that in Gibbs sampling, we sample one dimension at a time, while conditioning the distribution on all the other dimensions. In other words, we cycle through available dimensions and compute the conditional distribution in each iteration. That's exactly what the sample function does. For each sample, we iterate over each dimension and compute the mean and covariance of the Gaussian conditional distribution, from which we then take and record a sample. We repeat this process until the maximum number of samples is reached. Let's see the Gibbs sampling algorithm in action for sampling from a 2D Gaussian distribution.

**Listing 2.4 Gibbs sampling**

```
import numpy as np
import matplotlib.pyplot as plt
```

```
import itertools
from numpy.linalg import inv
from scipy.stats import multivariate_normal

np.random.seed(42)

class gibbs_gauss:

 def gauss_conditional(self, mu, Sigma, setA, x):
     dim = len(mu)
     setU = set(range(dim))
     setB = setU.difference(setA)
     muA = np.array([mu[item] for item in setA]).reshape(-1,1)
     muB = np.array([mu[item] for item in setB]).reshape(-1,1)
     xB = np.array([x[item] for item in setB]).reshape(-1,1)

     Sigma_AA = []
     for (idx1, idx2) in itertools.product(setA, setA):
         Sigma_AA.append(Sigma[idx1][idx2])
     Sigma_AA = np.array(Sigma_AA).reshape(len(setA),len(setA))

     Sigma_AB = []
     for (idx1, idx2) in itertools.product(setA, setB):
         Sigma_AB.append(Sigma[idx1][idx2])
     Sigma_AB = np.array(Sigma_AB).reshape(len(setA),len(setB))

     Sigma_BB = []
     for (idx1, idx2) in itertools.product(setB, setB):
         Sigma_BB.append(Sigma[idx1][idx2])
     Sigma_BB = np.array(Sigma_BB).reshape(len(setB),len(setB))

     Sigma_BB_inv = inv(Sigma_BB)
     mu_AgivenB = muA + np.matmul(np.matmul(Sigma_AB, Sigma_BB_inv),
     ➥ xB - muB)
     Sigma_AgivenB = Sigma_AA - np.matmul(np.matmul(Sigma_AB,
     ➥ Sigma_BB_inv), np.transpose(Sigma_AB))

     return mu_AgivenB, Sigma_AgivenB

 def sample(self, mu, Sigma, xinit, num_samples):
     dim = len(mu)
     samples = np.zeros((num_samples, dim))
     x = xinit
     for s in range(num_samples):
         for d in range(dim):
             mu_AgivenB, Sigma_AgivenB = self.gauss_conditional(mu, Sigma,
             ➥ set([d]), x)
             x[d] = np.random.normal(mu_AgivenB, np.sqrt(Sigma_AgivenB))
         #end for
         samples[s,:] = np.transpose(x)
     #end for
     return samples

if __name__ == "__main__":

 num_samples = 2000
 mu = [1, 1]
```

Computes P(X_A | X_B = x) = N(mu_{A|B}, Sigma_{A|B})

```
Sigma = [[2,1], [1,1]]
xinit = np.random.rand(len(mu),1)
num_burnin = 1000

gg = gibbs_gauss()
gibbs_samples = gg.sample(mu, Sigma, xinit, num_samples)

scipy_samples =
➥ multivariate_normal.rvs(mean=mu,cov=Sigma,size=num_samples,random_state=42)

plt.figure()
plt.scatter(gibbs_samples[num_burnin:,0], gibbs_samples[num_burnin:,1],
➥ label='Gibbs Samples')
plt.grid(True); plt.legend(); plt.xlim([-4,5])
plt.title("Gibbs Sampling of Multivariate Gaussian"); plt.xlabel("X1");
➥ plt.ylabel("X2")
plt.show()

plt.figure()
plt.scatter(scipy_samples[num_burnin:,0], scipy_samples[num_burnin:,1],
➥ label='Ground Truth  Samples')
plt.grid(True); plt.legend(); plt.xlim([-4,5])
plt.title("Ground Truth Samples of Multivariate Gaussian");
➥ plt.xlabel("X1");
plt.ylabel("X2")
plt.show()
```

From figure 2.12, we can see that Gibbs samples resemble the ground truth 2D Gaussian distribution samples parameterized by $\mu = [1, 1]^T$ and $\Sigma = [[2, 1], [1, 1]]$. In the next section, we will examine a more general MCMC sampling algorithm: Metropolis-Hastings sampling.

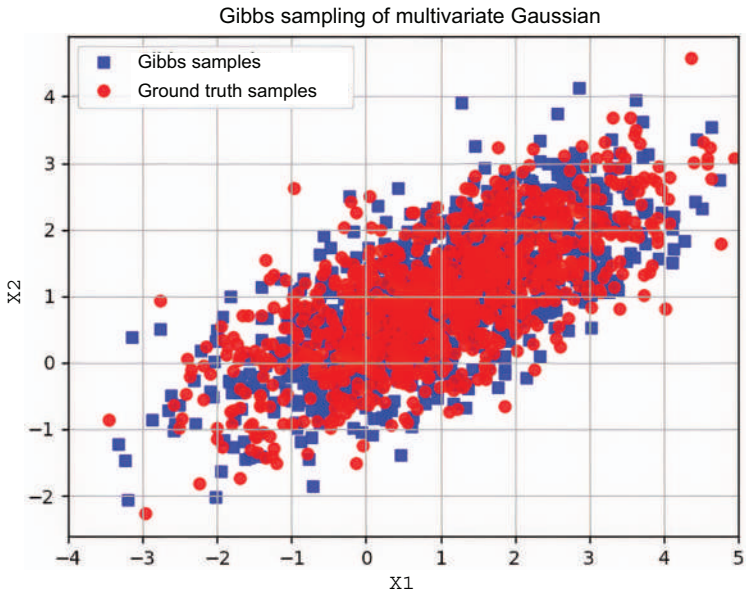


**Figure 2.12 Gibbs samples of a multivariate Gaussian**

## 2.6 Metropolis-Hastings sampling

Let's look at a more general MCMC algorithm for sampling from distributions. Our goal is to construct a Markov chain whose stationary distribution is equal to our target distribution $p(x)$. The target distribution $p(x)$ is the distribution (typically a posterior $p(\theta|x)$ or a density function $p(\theta)$) we are interested in drawing samples from.

The basic idea in the Metropolis-Hastings (MH) algorithm is to propose a move from the current state $x$ to a new state $x'$ based on a proposal distribution $q(x'|x)$, and then either accept or reject the proposed state according to MH ratio that ensures that detailed balance is satisfied, as shown in equation 2.18.

$$p(x')\, q(x|x') = p(x) q(x'|x) \tag{2.18}$$

The detailed balance equation states that the probability of transitioning out of state $x$ is equal to the probability of transitioning into state $x$. To derive the MH ratio, assume for a moment that the preceding detailed balanced equation is not satisfied, and then there must exist a correction factor $r(x'|x)$ such that the two sides are equal, and solving for it leads to the following MH ratio.

$$\begin{aligned} p(x')\, q(x|x') &= r(x'|x)\, p(x) q(x'|x) \\ r(x'|x) &= \min\left[1, \frac{p(x')\, q(x|x')}{p(x) q(x'|x)}\right] \end{aligned} \tag{2.19}$$

We can summarize the Metropolis-Hastings algorithm as shown in figure 2.13.

**Sample from the proposal distribution**

1: Init $x_0$ at random
2: **for** $k = 0, 1, 2, \ldots$ **do**
3: propose a new state $x' \sim q(x'|x_k)$
4: compute metropolis-hastings ratio:
5: $r(x'|x) = \min\left[1, \frac{p(x')q(x|x')}{p(x)q(x'|x)}\right]$
6: set $x_{k+1} = \begin{cases} x', & \text{with prob. } r(x'|x) \\ x_k, & \text{with prob. } 1 - r(x'|x) \end{cases}$ ← **Accepts the sample** / ← **Rejects the sample**
7: **end for**

**Figure 2.13 Metropolis-Hastings pseudo-code**

Let's implement the MH algorithm for a multivariate mixture of Gaussian target distribution and a Gaussian proposal distribution.

$$\begin{aligned} p(x) &= \sum_{k=1}^{K} \pi(k) \mathcal{N}(x; \mu_k, \Sigma_k) \\ q(x'|x) &= \mathcal{N}(x'|x, \Sigma) \end{aligned} \tag{2.20}$$

Let's examine code listing 2.5. The `mh_gauss` class consists of the `target_pdf` function that defined the target distribution (in our case, the Gaussian mixture $p(x)$); the `proposal_pdf` function that defines the proposal distribution (a multivariate normal); and the `sample` function, which samples a new state from the proposal $q(x'|xk)$ conditioned on the previous state $xk$, computes the Metropolis-Hastings ratio, and either accepts the new sample with probability $r(x'|x)$ or rejects the sample with probability $1 - r(x'|x)$.

**Listing 2.5 Metropolis-Hastings sampling**

```
import numpy as np
import matplotlib.pyplot as plt

from scipy.stats import uniform
from scipy.stats import multivariate_normal

np.random.seed(42)

class mh_gauss:

    def __init__(self, dim, K, num_samples, target_mu, target_sigma,
    ➥ target_pi, proposal_mu, proposal_sigma):
     self.dim = dim
     self.K = K
     self.num_samples = num_samples
     self.target_mu = target_mu
     self.target_sigma = target_sigma
     self.target_pi = target_pi

     self.proposal_mu = proposal_mu
     self.proposal_sigma = proposal_sigma

     self.n_accept = 0
     self.alpha = np.zeros(self.num_samples)
     self.mh_samples = np.zeros((self.num_samples,
     ➥ self.dim))

 def target_pdf(self, x):
     prob = 0
     for k in range(self.K):
         prob += self.target_pi[k]*\
             multivariate_normal.pdf(x,self
             ➥ .target_mu[:,k],self.target_sigma[:,:,k])
     #end for
     return prob

 def proposal_pdf(self, x):
        return multivariate_normal.pdf(x, self
        ➥ .proposal_mu, self.proposal_sigma)

 def sample(self):
     x_init = multivariate_normal
     ➥ .rvs(self.proposal_mu, self.proposal_sigma, 1)
     self.mh_samples[0,:] = x_init

     for i in range(self.num_samples-1):
         x_curr = self.mh_samples[i,:]
```

**Target parameters:**
**p(x) = \sum_k pi(k) N(x; mu_k, Sigma_k)**

**Proposal parameters:**
**q(x) = N(x; mu, Sigma)**

**Sampling chain params**

**Target pdf: p(x) = \sum_k pi(k) N(x; mu_k,Sigma_k)**

**Proposal pdf:**
**q(x) = N(x; mu, Sigma)**

**Draws the initial sample from the proposal.**

```
            x_new = multivariate_normal.rvs(x_curr, self
            ➥ .proposal_sigma, 1)

            self.alpha[i] = self.proposal_pdf(x_new) /
            ➥ self.proposal_pdf(x_curr)
            self.alpha[i] = self.alpha[i] *
            ➥ (self.target_pdf(x_new)/self.target_pdf(x_curr))

            r = min(1, self.alpha[i])
            u = uniform.rvs(loc=0, scale=1, size=1)
            if (u <= r):
                self.n_accept += 1
                self.mh_samples[i+1,:] = x_new
            else:
                self.mh_samples[i+1,:] = x_curr
        #end for
        print("MH acceptance ratio: ", self.n_accept/float(self.num_samples))

if __name__ == "__main__":

    dim = 2
    K = 2
    num_samples = 5000
    target_mu = np.zeros((dim, K))
    target_mu[:,0] = [4,0]
    target_mu[:,1] = [-4,0]
    target_sigma = np.zeros((dim, dim, K))
    target_sigma[:,:,0] = [[2,1],[1,1]]
    target_sigma[:,:,1] = [[1,0],[0,1]]
    target_pi = np.array([0.4, 0.6])

    proposal_mu = np.zeros((dim,1)).flatten()
    proposal_sigma = 10*np.eye(dim)


    mhg = mh_gauss(dim, K, num_samples, target_mu, target_sigma, target_pi,
    ➥ proposal_mu, proposal_sigma)
    mhg.sample()

    plt.figure()
    plt.scatter(mhg.mh_samples[:,0], mhg.mh_samples[:,1], label='MH samples')
    plt.grid(True); plt.legend()
    plt.title("Metropolis-Hastings Sampling of 2D Gaussian Mixture")
    plt.xlabel("X1"); plt.ylabel("X2")
    plt.show()
```

MH ratio

MH acceptance probability

Accept

Reject

From figure 2.14, we can see that MH samples resemble the ground truth mixture of two 2D Gaussian distributions with means $\mu 1$ = [4,0], $\mu 2$ = [–4,0]; covariances Σ_1 = [[2,1],[1,1]] and Σ_2 = [[1,0],[0,1]]; and mixture proportions $\pi$ = [0.4,0.6].

Notice that we are free to choose any proposal distribution $q(x'|x)$ that makes the method flexible. A good choice of the proposal will result in a high sample acceptance rate.

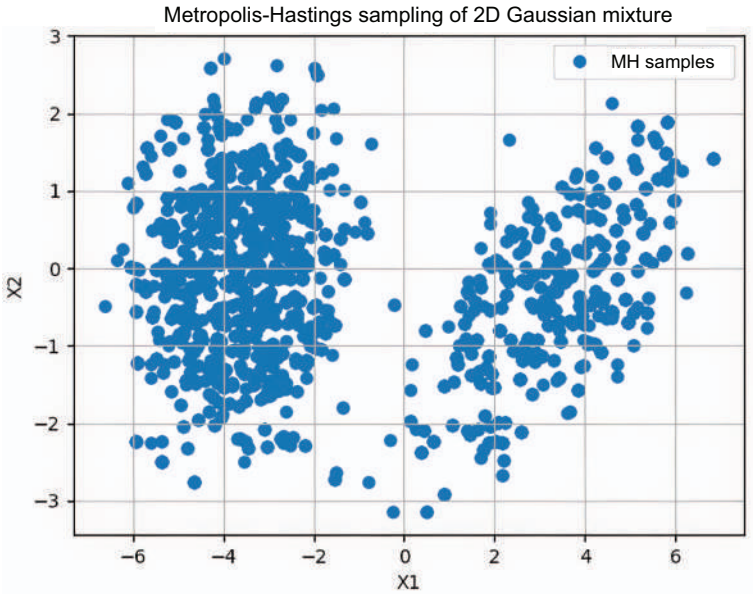


**Figure 2.14 Metropolis-Hastings samples of a multivariate Gaussian mixture**

In our implementation, we chose a symmetric Gaussian distribution centered on the current state shown in equation 2.21.

$$q\left(x'|x\right) = \mathcal{N}\left(x'|x, \Sigma\right) \tag{2.21}$$

This is known as a *random walk Metropolis algorithm*. If we use a proposal of the form $q(x'|x) = q(x')$, where the new state is independent of the old state, we get an independence sampler, which is similar to the importance sampling we will look at in the next section.

Also, notice that to compute the MH ratio, we don't need to know the normalization constants $Z$ of our distributions, since the ratio and the $Z$s cancel out. There's a connection between the MH algorithm and Gibbs sampling: the Gibbs algorithm acceptance ratio is always 1.

## 2.7 Importance sampling

*Importance sampling* (IS) is a Monte Carlo algorithm for estimating integrals of the following form.

$$E[f(x)] = \int p(x)f(x)dx \tag{2.22}$$

The idea behind importance sampling is to draw samples in interesting regions (i.e., where both $p(x)$ and $|f(x)|$ are large). Importance sampling works by drawing samples from an easier-to-sample proposal distribution $q(x)$. Thus, we can compute the expected value of $f(x)$ with respect to the target distribution $p(x)$ by drawing samples from the proposal $q(x)$ and using Monte Carlo integration.

$$E[f(x)] = \int f(x)\frac{p(x)}{q(x)}q(x)dx \approx \frac{1}{N}\sum_{i=1}^{N} w(x_i)f(x_i), \text{ where } x_i \sim q(x) \quad (2.23)$$

In equation 2.23, we defined the importance weights as $w(x) = p(x)/q(x)$. Let's look at an example where we use importance sampling to approximate an expected value of $f(x) = 2\sin(\pi x/1.5),\ x \geq 0$. We are asked to take the expectation with respect to an unnormalized Chi distribution parameterized by a noninteger degree of freedom (DoF) parameter $k$.

$$p(x) \sim x^{(k-1)} \exp\left\{-\frac{x^2}{2}\right\}, x \geq 0 \quad (2.24)$$

We will use an easier-to-sample from proposal distribution $q(x)$:

$$q(x) \sim \mathcal{N}(x; 0.8, 1.5) \quad (2.25)$$

Let's look at the pseudo-code in figure 2.15.

```
1: class importance_sampler
2: function sample(N):
3: for i = 1 to N:
4:     x_i ~ q(x) = N(x; μ, σ^2) ← Samples from the proposal
5:     w(x_i) = p(x_i)/q(x_i) ← Computes importance weights
6: end for
7: E[f(x)] = (1/N) Σ_{i=1}^{N} w(x_i) f(x_i)
8: return w(x), E[f(x)]
```

**Figure 2.15 Importance sampler pseudo-code**

Our `importance_sampler` class contains a function called sample that takes the number of samples $N$ as its input. Inside the sample function, we loop over the number of samples, and for each iteration, we sample from the proposal distribution $q(x)$ and compute the importance weight as the ratio of target distribution $p(x)$ to the proposal distribution $q(x)$. Finally, we compute the Monte Carlo integral by weighting our function of interest $f(x)$ by the importance weights, summing over all samples and dividing by $N$. Let's look at the following code listing, which captures all the details.

**Listing 2.6 Importance sampling**

```
import numpy as np
import matplotlib.pyplot as plt

from scipy.integrate import quad
from scipy.stats import multivariate_normal

np.random.seed(42)

class importance_sampler:

 def __init__(self, k=1.5, mu=0.8, sigma=np.sqrt(1.5), c=3):
     self.k = k
     self.mu = mu
     self.sigma = sigma
     self.c = c

 def target_pdf(self, x):
     return (x**(self.k-1)) * np.exp(-x**2/2.0)

 def proposal_pdf(self, x):
     return self.c * 1.0/np.sqrt(2*np.pi*1.5) *
     ➥ np.exp(-(x-self.mu)**2/(2*self.sigma**2))

 def fx(self, x):
        return 2*np.sin((np.pi/1.5)*x)

 def sample(self, num_samples):
     x = multivariate_normal.rvs(self.mu,
     ➥ self.sigma, num_samples)

     idx = np.where(x >= 0)
     x_pos = x[idx]

     isw = self.target_pdf(x_pos) /
     ➥ self.proposal_pdf(x_pos)

     fw = (isw/np.sum(isw))*self.fx(x_pos)
     f_est = np.sum(fw)

     return isw, f_est


if __name__ == "__main__":

 num_samples = [10, 100, 1000, 10000, 100000, 1000000]

 F_est_iter, IS_weights_var_iter = [], []
 for k in num_samples:
     IS = importance_sampler()
     IS_weights, F_est = IS.sample(k)
     IS_weights_var = np.var(IS_weights/np.sum(IS_weights))
     F_est_iter.append(F_est)
     IS_weights_var_iter.append(IS_weights_var)

 #ground truth (numerical integration)
 k = 1.5
```

Target parameters p(x)

Proposal parameters q(x)

Fixes c, s.t. p(x) < c q(x)

p(x) ~ Chi(k=1.5)

q(x) ~ N(mu,sigma)

Function of interest f(x), x >= 0

Sample from the proposal

Discards negative samples, since f(x) is defined for x >= 0

Computes importance weights

Computes E[f(x)] = sum_i f(x_i) w(x_i), where x_i ~ q(x)

```
I_gt, _ = quad(lambda x: 2.0*np.sin((np.pi/1.5)*x)*(x**(k-1))*np
➥ .exp(-x**2/2.0), 0, 5)

#generate plots
plt.figure()
xx = np.linspace(0,8,100)
plt.plot(xx, IS.target_pdf(xx), '-r', label='target pdf p(x)')
plt.plot(xx, IS.proposal_pdf(xx), '-b', label='proposal pdf q(x)')
plt.plot(xx, IS.fx(xx) * IS.target_pdf(xx), '-k', label='p(x)f(x)
    integrand')
plt.grid(True); plt.legend(); plt.xlabel("X1"); plt.ylabel("X2")
plt.title("Importance Sampling Components")
plt.show()

plt.figure()
plt.hist(IS_weights, label = "IS weights")
plt.grid(True); plt.legend();
plt.title("Importance Weights Histogram")
plt.show()

plt.figure()
plt.semilogx(num_samples, F_est_iter, label = "IS Estimate of E[f(x)]")
plt.semilogx(num_samples, I_gt*np.ones(len(num_samples)), label = "Ground
➥ Truth")
plt.grid(True); plt.legend(); plt.xlabel('iterations'); plt.ylabel("E[f(x)]
➥ estimate")
plt.title("IS Estimate of E[f(x)]")
plt.show()
```

Figure 2.16 (left) shows the target pdf $p(x)$, the proposal pdf $q(x)$, and the integrand $p(x)f(x)$. Notice that we scaled the proposal pdf $q(x)$ by constant $c$, such that $p(x) < cq(x)$ for all $x \geq 0$. Furthermore, we restricted the samples from $q(x)$ to be positive (thus, sampling from a truncated Gaussian) to meet our constraint of $x \geq 0$ for $f(x)$. Figure 2.16 shows the improvement in the estimate of $E[f(x)]$ vs. the number of samples compared to the ground truth computed via numerical integration.

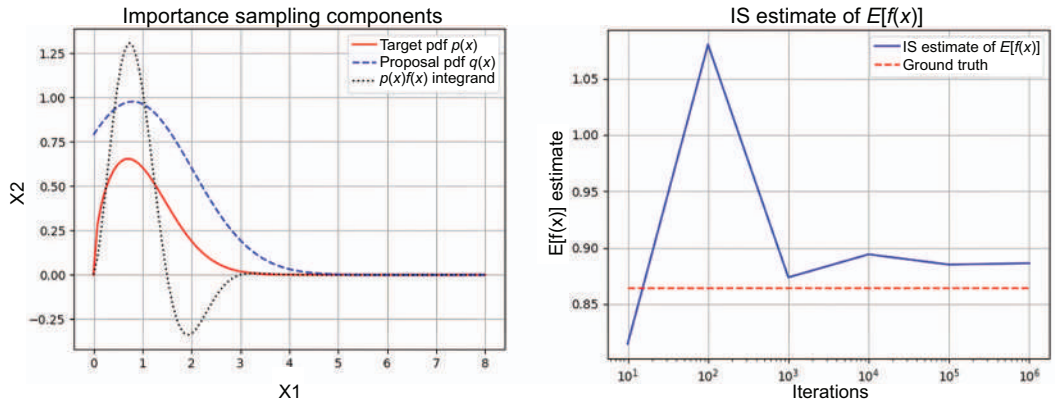


**Figure 2.16 IS components (left) and IS estimate (right)**

In the previous example, we used a normalized proposal distribution. However, for more complex distributions we often do not know the normalization constant (aka the partition function), since we are working with ratios of pdfs we'll see that the math holds for unnormalized distributions in addition we'll look at a way of estimating ratios of partition functions and discuss IS properties as an estimator.

Let's define a normalized PDF $p(x)$ and $q(x)$ as follows.

$$p(x) = \frac{1}{Z_p}\tilde{p}(x) \qquad q(x) = \frac{1}{Z_p}\tilde{q}(x) \tag{2.26}$$

Here, $Z_p$ and $Z_q$ are normalization constants of $p(x)$ and $q(x)$, respectively, and $\tilde{p}(x)$ and $\tilde{q}(x)$ are unnormalized distributions. We can write our estimate as follows.

$$\begin{aligned} E[f(x)] &= \int f(x)p(x)dx \\ &= \int f(x)\frac{q(x)}{q(x)}p(x)dx = \frac{Z_q}{Z_p}\int f(x)\frac{\tilde{p}(x)}{\tilde{q}(x)}q(x)dx \\ &\approx \frac{Z_q}{Z_p}\frac{1}{S}\sum_{s=1}^{S}\tilde{w}(s)f(s), \text{ where } s \sim q(x) \text{ and } \tilde{w}(s) = \frac{\tilde{p}(s)}{\tilde{q}(s)} \end{aligned} \tag{2.27}$$

Notice in equation 2.27, $\tilde{w}(s)$ are unnormalized importance weights. We can compute the ratio of normalizing constants as follows.

$$\frac{Z_p}{Z_q} = \frac{1}{Z_q}\int \tilde{p}(x)dx = \frac{1}{Z_q}\int \frac{\tilde{p}(x)}{q(x)}q(x)dx = \int \frac{\tilde{p}(x)}{\tilde{q}(x)}q(x)dx = \frac{1}{S}\sum_{s=1}^{S}\tilde{w}(s) \tag{2.28}$$

Combining the two expressions above, we get the following.

$$E[f(x)] \approx \frac{Z_q}{Z_p}\frac{1}{S}\sum_{s=1}^{S}\tilde{w}(s)f(s) = \frac{\frac{1}{S}\sum_s \tilde{w}(s)f(s)}{\frac{1}{S}\sum_s \tilde{w}(s)} = \sum_{s=1}^{S} w(s)f(s) \tag{2.29}$$

Equation 2.29 justifies our computation in the code example for the estimate of $E[f(x)]$. Let's look at a few properties of IS estimator and compute quantities relevant to characterizing the performance of IS. The IS estimator can be defined as follows.

$$\hat{a} = E[a(x)] = \frac{\frac{1}{n}\sum_{i=1}^{n} \tilde{w}^i a(x_i)}{\frac{1}{n}\sum_{i=1}^{n} \tilde{w}^i} = \frac{1}{n}\sum_{i=1}^{n} w_*^i a(x_i),$$
$$\text{where } w_*^i = \frac{\tilde{w}^i}{\frac{1}{n}\sum_i \tilde{w}^i} \tag{2.30}$$

By the weak law of large numbers (WLLN), assuming independent samples $x_i$, we know that the average of iid samples converges in probability to the true mean.

$$a = E[w_*(X_i)\, a(X_i)] \rightarrow b(\hat{a}) = E[\hat{a}] - a = 0 \tag{2.31}$$

Here, $X_i$ are assumed to be iid random variables distributed according to $q(x)$. As a result, IS is, in theory, an unbiased estimator. The variance of the IS estimator is given by the following.

$$\text{VAR}(\hat{a}) = \frac{\sigma^2}{n} = \frac{1}{n}\frac{\sum_{i=1}^{n}\left(\tilde{w}^i(a(x_i) - \hat{a})\right)^2}{\left(\sum_{i=1}^{n}\tilde{w}^i\right)^2} \tag{2.32}$$

To check whether the IS estimate is consistent, we need to show that as we increase the number of samples $n$, the estimate converges to the true value in probability.

$$\lim_{n\rightarrow\infty} P(|\hat{a}_n - a| > \epsilon) = 0 \tag{2.33}$$

Using Chebyshev's inequality, we can bound the deviation as follows.

$$\lim_{n\rightarrow\infty} P(|\hat{a} - a| > \epsilon) \leq \frac{\text{VAR}(\hat{a})}{\epsilon^2} = \frac{\sigma^2}{n\epsilon^2} = 0 \tag{2.34}$$

As a result, the IS estimator is consistent; here, we didn't need to use an assumption that samples are independent, but we needed to assume that the variance of importance weights is finite. Thus, we expect the variance around the estimate to shrink as we add more samples. One factor that affects the variance of the estimator is the magnitude of importance weights. Since $w(x_i) = p(x_i)/q(x_i)$, the weights are small if $q(x_i)$ is similar to p(xi) and has heavy tails. This is the desired case that leads to a small variance in our estimate.

We would like to have a way of knowing when the importance weights are problematic (e.g., when only a couple of weights dominate the weighted sum). When samples are correlated, the variance is $\sigma^2/n_{eff}$, where $n_{eff}$ is the effective sample size. To compute an expression for effective sample size, we set the variance of the weighted average equal to the variance of the unweighted average.

$$\text{VAR}\left(\frac{1}{n}\sum_{i=1}^{n} a(x_i)\right) = \frac{\sigma^2}{n_{eff}} = \text{VAR}\left(\frac{\frac{1}{n}\sum_{i=1}^{n} \tilde{w}^i a(x_i)}{\frac{1}{n}\sum_{i=1}^{n} \tilde{w}^i}\right)$$
$$\text{where VAR}(a(x_i)) = \sigma^2 \tag{2.35}$$

Solving for $n_{eff}$, we get the following equation.

$$n_{eff} = \frac{\left(\sum_{i=1}^{n} w_i\right)^2}{\sum_{i=1}^{n} w_i^2} = \frac{n}{1 + \text{VAR}\left(w_*^i\right)} \tag{2.36}$$

We can use $n_{eff}$ as a kind of diagnostic for the importance sampler, where the target number $n_{eff}$ depends on the application. The applications for importance sampling extend beyond approximating linear estimates, such as expectations and integrals as well as nonlinear estimates. In signal processing, importance sampling can be used for signal recovery, with fewer measurements.

## 2.8 Exercises

**2.1** Derive full conditionals $p(x_A | x_B)$ for a multivariate Gaussian distribution where $A$ and $B$ are subsets of $x_1, x_2, \ldots, x_n$ of jointly Gaussian random variables.

**2.2** Derive marginals $p(x_A)$ and $p(x_B)$ for a multivariate Gaussian distribution where $A$ and $B$ are subsets of $x_1, x_2, \ldots, x_n$ of jointly Gaussian random variables.

**2.3** Let $y \sim N(\mu, \Sigma)$, where $\Sigma = \text{LL}^{\text{T}}$. Show that you can get samples $y$ as follows: $x \sim N(0, I)$; $y = \text{L}x + \mu$.

## Summary

- Markov chain Monte Carlo (MCMC) is a methodology of sampling from high dimensional parameter spaces to approximate the posterior distribution $p(\theta|x)$.
- Monte Carlo (MC) integration has an advantage over numerical integration (which evaluates a function at a fixed grid of points), in that the function is only evaluated in places where there is a nonnegligible probability.
- In a binomial tree model of a stock price, we assume that at each time step, the stock could be in either up or down states with unequal payoffs characteristic of a risky asset.
- Self-avoiding random walks are simply random walks that do not cross themselves. We can use Monte Carlo integration to simulate a self-avoiding random walk.
- Gibbs sampling is based on the idea of sampling one variable at a time from a multidimensional distribution conditioned on the latest samples from all the other variables.

- The basic idea of the Metropolis-Hastings (MH) algorithm is to propose a move from the current state $x$ to a new state $x'$ based on a proposal distribution $q(x'|x)$ and then either accept or reject the proposed state according to MH ratio that ensures detailed balance is satisfied.
- The idea of importance sampling is to draw samples in interesting regions (i.e., where both $p(x)$ and $|f(x)|$ are large). Importance sampling works by drawing samples from an easier-to-sample proposal distribution $q(x)$.