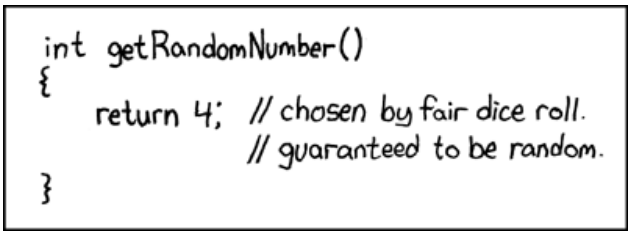
BRYN MAWR COLLEGE

Department of Physics

Computational Module 12 – Monte Carlo Methods

Prerequisite modules: Module 1, Module 3 – Iterative Methods
Estimated completion time: 3-6 hours
Learning objectives: Become familiar with computer implementation of random numbers; learn the Monte Carlo method of integration and about Monte Carlo simulation, especially its application to the classic Ising model



A legitimate random number generator?
Image credit: xkcd.com

Some situations in which a system is to be modeled involve parameters that are not well known, or that might vary over time in unpredictable ways. A nonphysical example is the financial markets: imagine trying to predict how much a pool of investments (stocks, bonds, mutual funds, etc.) might be worth a few decades from now without knowing how the stock and bond markets will perform during that time. One way to get some idea of the likely future value of the pool would be to run many simulations, each involving different assumptions about the performance of the markets, and then statistically analyze the set of predicted future values. This example illustrates one common application of ***Monte Carlo methods***, which is the topic of this Module. (The methods get their name from the "Monte Carlo" region of Monaco, home to the famous Monte Carlo casino – such establishments symbolize randomness, to the chagrin of gamblers!) Hereafter, the phrase "Monte Carlo" often will be abbreviated as MC.[1]

[1] This Module adapted from *Computational Physics*, by Mark Newman.

12.1 Random Numbers in Numerical Computation

While the physical world is full of truly random—that is, *nondeterministic*—phenomena (especially in the quantum realm), there currently is no known way of generating truly random numbers by computer. (That may change with the advent of quantum computers.) That's because conventional computers work solely by executing algorithms, which are fundamentally deterministic: you give them a certain input, and they perform a fixed and well-defined sequence of operations that always will produce the same output for a given input. The poor computer has no free will.

That said, most libraries of computational code include (deterministic) ***random number generators*** that produce what are termed ***pseudorandom numbers***. These are numbers that might appear random to the human eye, but that actually follow a pattern. (A series of numbers $x_1, x_2, ..., x_{n+1}$ would be truly random only if the probability of getting any particular one of the possible values for x_{n+1} was independent of the previous n values.) One such generator is the ***linear congruential random number generator***, which computes its output using the equation $x' = (ax + c) \bmod m$, where a, c and m are integer constants and x is an integer variable. A sequence of pseudorandom numbers can be obtained through an interative process: plug in a value for x on the right, compute the output x', use it as the new input on the right, and so on. Of course, if the process is repeated using the same values of the constants and the same initial x, the string of outputs will be identical. However, that string will be different if a different initial value of x, known as the ***seed*** for the generator, is used. The usual approach is to use the computer's current time and date values combined into one number as the seed value, since this seed never will repeat. It turns out that this generator function is a fairly bad one (the values produced are not independent of each other), so others are usually used for sensitive calculations.

A popular generator for physics problems is the ***Mersenne twister***, a complicated algorithm to program. However, Python has an implementation, which comes with the `random` library. This library contains the `random()` function (which takes no argument), which produces a *float* in the range $0 \le x < 1$ with *uniform probability* throughout the range. If a different upper limit is desired, one simply multiplies the output of `random` by that value; if a shifted range is needed, one just adds a constant to the output. (Of course, one can also do both, if necessary.) The `random` library also contains the `randrange` function, called as `randrange(m,n,k)`, which gives a random *integer* i in the range $m \le i \le n-1$ in steps of k. (If m or k are omitted, they default to 0 and 1, respectively.) These functions use the computer's system time as the seed by default, but if you want to use the same seed repeatedly (to check for reproducibility of code output), you can use the `seed` function in the library. Calling it as `seed(x)` sets `x` as the seed for later calls of the other functions.

Interestingly, Unix-based systems contain a file called `/dev/random` which collects random noise data from some of the host computer's internal devices as a source of entropy; the resulting random numbers generated in the file are regarded as very high quality, suitable for cryptographic uses. In this way, the computer can tap into the true randomness of the physical world.

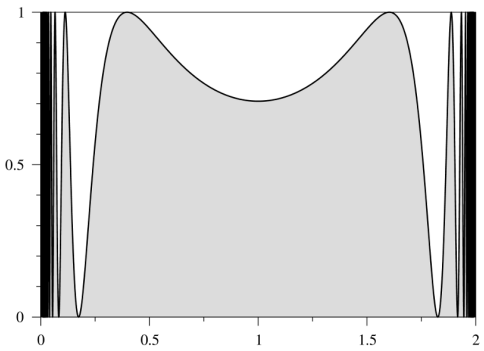


Figure 1: A pathological function

12.2 Monte Carlo Integration

One clever application of random numbers is in the computation of integrals, where the approach is known as ***Monte Carlo integration***. As a simple example, suppose you want to find the area of a disk of radius R (and you don't know the formula $A = \pi R^2$). We could adapt our integration methods from Module 5 to do this, but an alternative is to imagine circumscribing a square with sides of length R around the disk, so that the disk just touches the square at the centers of its sides. If we knew the ratio of areas of the disk and square, and that the area of the square is just R^2, then we could determine the area of the disk. We can compute approximate values of those areas by generating a lot of random numbers corresponding to points within the square, and counting how many of them fall within the disk. The ratio of the number inside the disk to the total number of points is an approximation to the ratio of areas. If our random number generator is good, and we use enough points, this method can give a decent result.

The error in a MC integration using N random points is $N^{-1/2}$, while the trapezoidal and Simpson's methods have errors of N^{-2} and N^{-4}, respectively. Thus, calculations using $N = 100$ would produce errors of 10% for the MC method, 0.01% for the trapezoidal method, and 0.000001% for Simpson's method. Despite this lack of accuracy of the MC method, it may be a superior approach in the case of very poorly-behaved ("pathological") functions; for example, the function $f(x) = \sin^2[1/x(2-x)]$ shown in Figure 1,[2] which oscillates increasingly rapidly as x approaches 0 and 2. Such a function would be very difficult if not impossible to integrate in those regions using traditional techniques.

12.2.1 The Mean Value Method

The Monte Carlo method described above is a particularly inaccurate one and there are, in fact, better approaches based on random numbers. A somewhat superior approach is the ***mean value method***. The idea here is that the integral (I) of a function $f(x)$ from $x = a$ to $x = b$ can be written as $I = (b-a)\langle f \rangle$, where $\langle f \rangle$ is the average of $f(x)$ in the range. The average value can

[2] Image from *Computational Physics* by Newman

be approximated by evaluating the function at N random points x_i in the range and taking the average of those values:

$$I = (b-a)\langle f \rangle \simeq (b-a)\frac{1}{N}\sum_{i=1}^{N} f(x_i) \tag{1}$$

It turns out that the error in this approach again goes as $N^{-1/2}$, but since the coefficient multiplying $N^{-1/2}$ is smaller in this case, the mean value method always is more accurate than the Monte Carlo one (i.e.; if the error in the MC approach is written as $aN^{-1/2}$ and that of the mean value method as $bN^{-1/2}$, then $b < a$, so the mean value method has smaller error).

12.2.2 Importance Sampling

In some situations, the straightforward MC integration approach will work poorly. An example is in the integral below, which comes up in statistical mechanics in the study of gases of fermions:

$$I = \int_0^1 \frac{x^{-1/2}}{e^x + 1}\, dx. \tag{2}$$

The integral is finite in value even though the integrand diverges at $x = 0$. But this divergence renders the MC integration approach unreliable: it is very sensitive to sample points x_i that come very close to 0, and a particularly close one can result in a spuriously large output. The way around this is to choose the sample points x_i *not* with uniform probability in the range $x = 0 \to 1$, but instead to use an appropriately weighted *nonuniform* distribution. This technique is known as ***importance sampling***.

The idea is to choose some ***weighting function*** $w(x)$, in terms of which we can define a ***weighted average*** of another function $g(x)$ over the interval from a to b as

$$\langle g \rangle_w = \frac{\int_a^b w(x)\, g(x)\, dx}{\int_a^b w(x)\, dx}. \tag{3}$$

(Here, if $w(x) = 1$ we get the standard result for the average. This expression also parallels what you might have seen for a weighted average as a sum rather than an integral.) To apply this to the problem of computing the integral $I = \int_a^b f(x)\, dx$, set $g(x) = f(x)/w(x)$ in Eq. (3) to get

$$\left\langle \frac{f(x)}{w(x)} \right\rangle_w = \frac{\int_a^b f(x)\, dx}{\int_a^b w(x)\, dx}, \tag{4}$$

from which we obtain

$$I = \int_a^b f(x)\, dx = \left\langle \frac{f(x)}{w(x)} \right\rangle_w \int_a^b w(x)\, dx. \tag{5}$$

We still need to specify how to compute the average term multiplying the integral on the right. That is done by introducing a normalized version of $w(x)$, which represents a ***probability density function*** that gives the probability of obtaining a value in the range $x \to x + dx$:

$$p(x) \equiv \frac{w(x)}{\int_a^b w(x)\,dx}. \tag{6}$$

(As should be clear, the integral of this quantity over the range of interest is 1.) We now use p to generate a nonuniform set of N sample points x_i by setting $p(x)\,dx$ as the probability of taking a sample point from the range $x \to x + dx$. The number of sample points in that range then will be $Np(x)\,dx$, and so for any arbitrary function $g(x)$ we can write the approximation

$$\sum_{i=1}^{N} g(x_i) \simeq \int_a^b N\,p(x)\,g(x)\,dx. \tag{7}$$

To understand this expression, consider an infinitesimal interval dx in which a small number of x_i lie. The sum of the $g(x_i)$ in that range is approximately equal to the number of x_i in that range times a typical value of $g(x)$ in the range; i.e., $\sum_{x_i \text{ in } dx} g(x_i) \simeq [Np(x)dx]\,g(x)$. Extending this approximation over the entire x range gives the expression above.

Looking back to Eq. (3), this lets us write the weighted average of $g(x)$ as

$$\langle g \rangle_w = \frac{\int_a^b w(x)g(x)\,dx}{\int_a^b w(x)\,dx} = \int_a^b p(x)\,g(x)\,dx \simeq \frac{1}{N}\sum_{i=1}^{N} g(x_i). \tag{8}$$

This looks just like the standard average, but remember that the points x_i are being chosen from the *nonuniform* distribution $p(x)$. If we use this result to rewrite Eq. (4), we obtain

$$\left\langle \frac{f(x)}{w(x)} \right\rangle_w \simeq \frac{1}{N}\sum_{i=1}^{N} \frac{f(x_i)}{w(x_i)}, \tag{9}$$

and finally, the integral of Eq. (5) becomes

$$I = \int_a^b f(x)\,dx \simeq \frac{1}{N}\sum_{i=1}^{N} \frac{f(x_i)}{w(x_i)} \int_a^b w(x)\,dx. \tag{10}$$

Note that this is a generalization of the mean value expression, Eq. (1): setting $w(x) = 1$, we recover that result. The idea now is to choose $w(x)$ to tame the troublesome parts of the function to be integrated. For example, in the case of the function of Eq. (2) we could choose $w(x) = x^{-1/2}$, in which case $f(x)/w(x) = (e^x + 1)^{-1}$, which is perfectly well-behaved in the range of interest.

12.2.3 The Transformation Method

The last issue to deal with is the fact that Python has no function for generating random numbers from a nonuniform distribution: `random()` is based on a uniform distribution. That problem can be solved, however, essentially by performing a change of variables.

Assume we have a function $x(z)$, as well as some general probability distribution $q(z)$, so the probability of generating a number in the range $z \to z + dz$ is $q(z)\,dz$. For every random number z there is a corresponding $x(z)$, but in general those numbers will follow some other distribution, $p(x)$. The trick, then, is to choose $x(z)$ so that $p(x)$ is the distribution we seek. (In other words, a proper choice of $x(z)$ will convert the initial distribution $q(z)$ to the desired one $p(x)$.) The way to do this is to note that the probability of getting a value of x in the range $x(z) \to x(z) + dx = x(z + dz)$, which is $p(x)\,dx$, is the same as the probability of getting a value of z in the corresponding z interval $(z \to z + dz)$, which is $q(z)\,dz$:

$$p(x)\,dx = q(z)\,dz. \tag{11}$$

As noted earlier, in the case of Python the distribution used by `random()` is uniform—$q(z) = 1$ over the interval from 0 to 1 and zero everywhere else—so if we integrate the two sides of the previous equation over part of that range, from 0 to z, we find

$$\int_{-\infty}^{x(z)} p(x')\,dx' = \int_0^z q(z')\,dz' = \int_0^z dz' = z. \tag{12}$$

Technically, the lower limit on the first integral should be $x(0)$, but since $q(z) = 0$ for $z < 0$, then $p(x) = 0$ for $x < x(0)$, so we can extend the lower limit of the x integral to anywhere below $x(0)$. The integral on the left will give some function of x, which must equal z.

There are two "catches" to this approach: (i) the integral on the left can't always be performed; and (ii) the resulting equation, which will be of the form $f(x) = z$, cannot always be inverted to obtain x as a function of z, i.e. $x(z)$. If it can, however, then we will have what we wanted: we obtain a set of z values from a uniform probability distribution (such as provided by `random()`), and the corresponding $x(z)$ values will represent the desired nonuniform distribution. This approach to constructing a nonuniform probability distribution is known as the ***transformation method***.

12.3 Monte Carlo Simulations

12.3.1 Markov Chain Monte Carlo Method

Monte Carlo methods often are used in physics, particularly in statistical mechanics, to simulate physical processes. The resulting simulations are referred to as ***Monte Carlo simulations***.

Using statistical mechanics as an example application, a common goal is to compute the average of

some quantity in a system (e.g., the pressure in a gas) in thermal equilibrium at temperature T (in kelvins). Statistical mechanics theory tells us that while we cannot say specifically what state the system would be in (what all the particle speeds would be in a gas, for instance), the probability of its being in a state with energy E_i is related to the "Boltzmann factor," $e^{-\beta E_i}$, via

$$P(E_i) = \frac{e^{-\beta E_i}}{Z}, \tag{13}$$

where $\beta \equiv 1/k_B T$ (k_B is "Boltzmann's constant") and $Z \equiv \sum_i e^{-\beta E_i}$ is the "partition function," involving a sum over all "microstates" of the system. The average value of some quantity X, whose value in the i^{th} state is X_i, then is

$$\langle X \rangle = \sum_i X_i \, P(E_i). \tag{14}$$

Usually, this sum cannot be determined analytically, so it must be computed numerically. However, it would be prohibitively time-consuming, if not impossible, to do it directly, as the number of states involved typically is astronomically large. MC techniques can come to the rescue here: the idea, similar to the approach we used before, is to replace the complete sum by one over a smaller number of terms chosen randomly. If we choose N of the states, and designate them by the subscript j, then we can write

$$\langle X \rangle \simeq \frac{\sum_{j=1}^{N} X_j P(E_j)}{\sum_{j=1}^{N} P(E_j)}, \tag{15}$$

where we have to include the denominator for normalization. It was not needed in the previous expression because the probability is normalized for the full sum; i.e., $\sum_i P(E_i) = 1$ when *all* states i are included.

It turns out that in this instance the simple approach of using a uniform sampling of states will give poor results: most of the states are associated with Boltzmann factors that are so small (since $E_j \gg k_B T$ for them) that they contribute a negligible amount to the sum. In fact, most of the sum comes from a very small subset of the terms, most or all of which are likely to be missed in a uniform random sample. Here's a situation where we clearly should use importance sampling!

Following a procedure similar to that by which Eq. (10) was derived for the integral $I = \int_a^b f(x)\, dx$, it can be shown that the average above is expressible, in terms of weights w_i that we choose, as

$$\langle X \rangle \simeq \frac{1}{N} \sum_{j=1}^{N} \frac{X_j P(E_j)}{w_j} \sum_i w_i, \tag{16}$$

where the last sum is over *all* of the states, and normally is computed analytically. [In essence, $X_j P(E_j)$—the quantity we need to sum in Eq. (14)—takes the place of $f(x_i)$ in Eq. (10).]

The aim now is to choose the weights w_i to pick out the large $P(E_i)$ terms in the sum, and such that the sum of the w_i can be determined analytically. But this is easy: we simply take $w_i = P(E_i)$ which, by Eq. (13), are normalized, $\sum_i w_i = \sum_i P(E_i) = 1$. Then, the previous expression becomes

$$\langle X \rangle \simeq \frac{1}{N} \sum_{j=1}^{N} X_j. \tag{17}$$

This is telling us that to average X, we just choose N states in proportion to their Boltzmann probabilities; i.e., their likelihood of occurring. (Of course, the more states we use, the more accurate the approximation.) This averaging process is a simulation of the actual physical process in that the system will find itself most often in the states having the highest probabilities, so an average of a quantity X in the system will be dominated by the values in the most probable states, just as in the average we've constructed.

There's a complication here: the probabilities $P(E_i)$ depend on the partition function Z, which we usually don't know *a priori* and which often is difficult to calculate numerically due to the large number of states involved, as mentioned earlier. It is, however, possible to choose the probabilities without knowing Z, using a powerful model for random processes known as a ***Markov chain***, named after Russian mathematician Andrei Markov. (See the associated Wikipedia page, which lists numerous applications, including to the modeling of chemical reactions, to speech recognition systems, macroeconomic and population genetics models, and even Google's PageRank algorithm.)

A Markov chain represents a random process in which a system transitions through a sequence of connected ***states*** (analogous to links in a chain), the sequence being determined by ***transition probabilities*** T_{ij} from state i to state j which are *independent of the previous states in the chain.* That is, in a Markov chain T_{ij} depends only on the current state i and the next state j. For this reason, the Markov chain is said to be "memoryless." (Not all chains are.)

In the present application, the states in the Markov chain will correspond to actual states of our physical system. For a long enough chain, the probability of *being in* a particular state will be the same as the probability of *transitioning to* that state. The idea, then, is to choose the T_{ij} such that the probability of transitioning into any particular state i on any step in the chain is the Boltzmann probability for that state, $P(E_i)$.

In order for the T_{ij} to satisfy this condition, we first note that the T_{ij} will be normalized according to $\sum_j T_{ij} = 1$, since the system must transition to *some* state from the i^{th} one. We then choose the T_{ij} so that they satisfy

$$\frac{T_{ij}}{T_{ji}} = \frac{P(E_j)}{P(E_i)} = \frac{e^{-\beta E_j}/Z}{e^{-\beta E_i}/Z} = e^{-\beta(E_j - E_i)}. \tag{18}$$

Note that the partition function Z has been eliminated from this result, so it will not need to be

calculated. To check whether this way of choosing the T_{ij} will work as we want, suppose that the probability that the system is in any state i at any step in the chain is $P(E_i)$; in the next step, the probability of the system being in state j will be the sum of the probabilities of getting there from all possible prior states i (those probabilities are the T_{ij}'s). That probability should be $P(E_j)$.

Breakpoint 1: Using the earlier expressions involving the T_{ij}'s, prove the claim made above—that $\sum_i P(E_i)\, T_{ij} = P(E_j)$.

The *Breakpoint* result demonstrates that the choice of T_{ij} implicit in Eq. (18) will, indeed, achieve our goal: if the probability of being in any particular state in the *first* step is what we want it to be, then the probability of being in any particular state in the *next* step also will be correct, and so on through the entire chain.

We have yet to actually determine the T_{ij} (as opposed to their ratios), so there's one more step to this process. In fact, there are many choices of the T_{ij} consistent with the ratio in Eq. (18), so we have considerable freedom to make the choice as we wish. It is conventional to make the choice so as to be consistent with the ***Metropolis algorithm*** (also known as the Metropolis-Hastings algorithm), named after Nicholas Metropolis. (The actual credit for this algorithm seems to be a matter of dispute: see the associated Wikipedia page for an interesting discussion. Note that physicist Edward Teller was involved in the algorithm's development.)

Now suppose that we start with the system in state i, and consider making a small change that would put the system in state j; e.g., by changing the energy of one gas particle. (Note that the new state could be the same as the previous state; i.e., $T_{ii} \neq 0$, in general. This would happen, for instance, if two gas particles collided elastically and interchanged energies.) The symbols i and j here designate steps in the Markov chain rather than particular states of the system.) The Metropolis algorithm instructs us to accept such a change with an ***acceptance probability***

$$P_a = \begin{cases} 1 & \text{if } E_j \leq E_i, \\ e^{-\beta(E_j - E_i)} & \text{if } E_j > E_i. \end{cases} \tag{19}$$

The idea here is that the system always will "choose" to go to a state with equal or lower energy than the current state, but it also may move to a state of higher energy, with some probability dependent on the temperature (through $\beta = 1/k_B T$). (If this didn't happen, numerical models of thermodynamic systems would always result in them evolving directly to their ground state which, in fact, doesn't happen due to thermal excitations.) In the second case above, the probability of transitioning to a state of higher energy is inversely related to the change in energy (the exponential factor becomes smaller as $E_j - E_i$ grows), and it's also inversely related to T (smaller T leads to a larger negative exponent and therefore a smaller exponential factor). This simulated behavior is designed to match what we expect to happen physically: it's more likely for our system to jump to

a higher energy if it's in a higher-temperature environment than a lower temperature environment, but the jump is less likely for a larger energy difference.

To convert the acceptance probability into a transition probability, suppose that the number of possible changes to the system is M. Then the probability of transitioning from state i to state j is the probability of choosing a particular change (which would be just $1/M$), times the probability that we accept that change. This lets us write the transition probabilities as

$$\begin{cases} T_{ij} = \dfrac{1}{M} \times e^{-\beta(E_j - E_i)} \quad \text{and} \quad T_{ji} = \dfrac{1}{M} \times 1, & \text{if } E_j > E_i; \\ T_{ij} = \dfrac{1}{M} \times 1 \quad \text{and} \quad T_{ji} = \dfrac{1}{M} \times e^{-\beta(E_i - E_j)}, & \text{if } E_j \leq E_i. \end{cases} \tag{20}$$

Breakpoint 2: Prove that these choices satisfy the relation shown in Eq. (18).

Now that we have identified all of the pieces, we can spell out the ***Markov Chain Monte Carlo*** (MCMC) simulation method:

1. Start in an arbitrary allowed initial state.
2. Select a change to the system at random from among the set of possible changes, and compute the change in energy from the previous state.
3. Compute the corresponding acceptance probability P_a from Eq. (19) using the energy change. Accept the change with probability P_a—i.e., with that probability change to the new state; otherwise, reject the change.
4. Compute and store the value of the quantity (or quantities) of interest in the current state.
5. Repeat from step 2.

Step 3 of the sequence above can be implemented by using `random()` to obtain a random number z and comparing it with $e^{-\beta(E_j - E_i)}$; if $z < e^{-\beta(E_j - E_i)}$, then the change to the new state is made; if not, the change is rejected and the system remains in the same state. (The larger $e^{-\beta(E_j - E_i)}$ is, the more likely z is to be less than it and the more likely the change is to be accepted.)

It's important to emphasize a few points about the MCMC approach. First, even if a change of state is rejected and the system remains in its current state, that still counts as a "step" in the Markov chain, and therefore the value of any quantity of interest that's being recorded should be recorded again (and/or again added to any sum of its values). Second, in order for Eq. (18) to hold, the number of changes to the system that are available in going from state i to state j must be the same number that would be available in making the reverse change. This might require careful selection of the set of possible state changes. Third, it must be possible eventually to get to any

state from any other state; this property of the system, known as ***ergodicity***, is a requirement for the Metropolis algorithm to work. Finally, while the Markov chain process always will converge to the correct Boltzmann probability distribution, there's no way to predict how many steps that will take; i.e., how long it will take for the system to ***equilibrate***. Basically, one determines when equilibration has been achieved by looking for minimal or no changes in the output values.

12.3.2 The Ising Model

A classic physical system to which the MCMC method has been applied is the ***Ising model***, used to represent the behavior of a magnet. We imagine a regular, square 2-D lattice (one can extend the model to 3-D) of magnetic dipoles ("spins") which, for simplicity, we assume can point only "up" or "down." The degree of alignment of the many individual dipoles determines the overall magnetization of the magnet.

In many real magnetic materials, pairs of dipoles interact in a way proportional to the dot product of their quantum-mechanical spin vectors, $\mathbf{s}_i \cdot \mathbf{s}_j$; in the Ising model this interaction is simplified by assuming an interaction energy (E_{ij}) between spins s_i and s_j proportional to the product of the up ($s = +1$) or down ($s = -1$) spins; that is, $E_{ij} \propto s_i s_j$. More specifically, the energy of interaction of two spins in a ferromagnetic material (in which neighboring spins like to be aligned) is written as $E_{ij} = -J s_i s_j$, where J is a positive interaction constant called the ***interaction energy***. (J would be negative for an *anti*ferromagnetic material.) This energy expression makes sense: if s_i and s_j have the same sign, meaning they're aligned, then the energy is negative and that state is relatively easy to attain, whereas if they have opposite signs then the energy is positive and that higher-energy state is harder for the system to reach. (Opposite statements hold for an antiferromagnetic material.)

Assuming, as is normally done, that the dipoles interact only with their nearest neighbors (the ones directly above, below, left, and right), then the total interaction energy of the magnet is

$$E = -J \sum_{\langle ij \rangle} s_i s_j, \tag{21}$$

where $\langle ij \rangle$ represents all nearest-neighbor pairs of spins. (Double-counting of the spin pairs must be avoided in computing this sum.)

Below is a Python function to implement the MCMC approach to the solution of the 2-D Ising model. It allows the user to input the size of the lattice, the constant J, and the temperature T. The main lattice of spins (occupying rows and columns lying in the index range $1 \rightarrow N$) is augmented with a border of values (with row/column indices 0 and $N + 1$) that impose ***periodic boundary conditions***: the elements in row 1 are duplicated in row $N + 1$, those in row N are duplicated in row 0, and likewise for the columns. These boundary conditions allow sensible computations even with lattice elements adjacent to the border (i.e., those with row/column indices of 1 or N).

The idea is to start with the system in some random arrangement, and then to consider "flipping" one of the spins. That flip will be accepted or rejected, based on the MCMC algorithm. After this is done many times (with many different spins), the system will have evolved to some steady state whose average properties we're interested in.

While, in principle, we are supposed to calculate *all* of the pairwise interactions throughout the lattice in each step of the process, that is unnecessary in this situation: when we flip one spin, only four interaction energies change—in fact, they become just the negatives of their previous values—so we don't need to compute all of the other interaction energies. We just compute the interaction energies of the spin we're considering flipping with its four nearest neighbors, both before and after the flip, and use the change in this "local" energy in the acceptance probability calculation. No other interaction energies will change, so they are irrelevant.

Models like this one allow for deep investigation of magnetic systems and give results for parameters (magnetization, energy, heat capacity) that, for large enough lattices, are in good agreement with approximate theoretical predictions and with observations of actual magnetic systems. ("Large enough" may be as small as 40×40). Here's code to do the simulation:

```
def isingPB(N, T, J):
    '''Analyze the 2-D Ising model using the Markov Chain Monte Carlo method.
    The user inputs the number of spins along one of the two (equal)
    dimensions, "N", the temperature "T", and the interaction constant "J".
    Periodic boundary conditions are used.'''

    from numpy import zeros, exp
    from numpy.random import choice, random

    kB = 1.       # Boltzmann constant in special units

    # Create array to hold spin values, which can be +/- 1
    # The N x N array will be surrounded by a "border" of values that impose
    # periodic boundary conditions.
    latt = zeros([N+2,N+2])
    latt[1:N+1,1:N+1] = choice([-1,1],[N,N])   # array of random +/- 1 values

    # Function to impose periodic boundary conditions on lattice
    # The first and last rows and columns store the wraparound values
    def imposeBCs():
        latt[0,1:N+1] = latt[N,1:N+1]
        latt[N+1,1:N+1] = latt[1,1:N+1]
        latt[1:N+1,0] = latt[1:N+1,N]
        latt[1:N+1,N+1] = latt[1:N+1,1]

    imposeBCs()                                # call the function above

    cntr = 0
```

```
    while cntr < 1000000:

        # Randomly choose a spin to flip
        row, col = choice(range(1,N+1),[2])    # pair of random indices

        # Local energy before spin flip
        oldE = -J * latt[row,col] * (latt[row-1,col] + latt[row+1,col] + \
               latt[row,col-1] + latt[row,col+1])

        # Local energy of potential new state will be the negative of
        # that of the original state, so
        deltaE = -2. * oldE                    # energy change

        # Determine whether to accept state change
        acceptval = random()    # random value for acceptance prob. comparison
        boltzfactor = exp(-deltaE/(kB * T))
        if acceptval < boltzfactor:            # accept state change
            latt[row, col] = -latt[row, col]   # flip spin
            imposeBCs()                        # reimpose boundary conditions

        cntr += 1

    mag = latt[1:N+1, 1:N+1].sum()     # sum spins to get net magnetization

    return mag
```

12.3.3 Simulated Annealing

A relatively recent development in the world of numerical methods is a technique for finding the ***global minimum*** of a function—the lowest of *all* possible values. Numerical techniques for finding a ***local minimum*** (the lowest value within some limited region) have been around for a long time and work reliably, but they are not guaranteed of finding a global minimum. The technique of ***simulated annealing*** (SA), published in 1983 by physicist Scott Kirkpatrick (and others) and based on statistical mechanics, provides a means of finding the global minimum of a function.

The SA idea stems from consideration, again, of a physical system in equilibrium at temperature T. Suppose that its ground-state (i.e., minimum) energy is $E_0 = 0$ and that all other states have positive energy, and recall that the Boltzmann probability for being in any particular state is $P(E_i) = e^{-\beta E_i}/Z$. Then, as $T \to 0$, $P(E_0) \to 1$, while $P(E_i) \to 0$ for any other state. That is, taking the limit $T \to 0$ leads the system to its ground state—the state in which its energy function is minimized. This agrees with what was described for the behavior of the Ising model as the temperature falls below the Curie temperature.

This kind of process can be visualized by imagining the energy function of the system as a landscape, and the state of the system at any moment as represented by a ball rolling or bouncing on the

landscape. The temperature T can be thought of as characterizing the "jitteriness" of the ball's motion: at high T it jitters and bounces around a lot, and so it would not be unusual to find it high up in the landscape (i.e., having high energy); at low T that jitteriness is small and so the ball is likely to have settled down into a low point (at least a local one) of the landscape.

Kirkpatrick's "aha moment" was to realize that the same approach that works in the statistical mechanics context to find the minimum energy could be used to find the minimum of *any* (reasonably well-behaved) function $f(x_1, x_2, \dots)$: by making small changes to the variables $x_1, x_2, \dots$, performing a MCMC simulation using f as the energy of the system, and taking the limit $T \to 0$. (A function f can be *maximized* by treating $-f$ as the energy.)

There's just one problem. If the system gets caught in a *local* minimum when T is low, then it's unlikely to get out: the acceptance probability would be very small for a change of state that would result in an energy gain large enough to get the system out of the local minimum (as $T \to 0$, $e^{-\beta(E_j - E_i)} \to 0$ for $E_j > E_i$). There is a solution to this problem, and it comes from actual physical processes; particularly glassblowing, metalworking, and similar activities. Workers in those areas know that if hot material is cooled too quickly, then the material can solidify with embedded imperfections or "defects": irregularities in the molecular structure of the material resulting from giving them insufficient time to settle into more highly-ordered configurations. Essentially, the molecules get trapped in local energy minima before they can find the globally minimum energy arrangement (without defects). Materials that form with these defects are more brittle than they should be. The solution is to cool the hot material slowly enough so that the molecules have time to find the globally minimum energy arrangement. In this case, if they happen to fall into a local energy minimum, the temperature may remain high enough for long enough that they have a good chance of escaping the local minimum and continuing their evolution toward the global one. It is this process of slow cooling that is referred to as simulated annealing.

The way in which the temperature is lowered in these models is referred to as a ***cooling schedule***. There is no single cooling schedule that will be optimal for all problems, but a common first try is an exponential function

$$T = T_0 e^{-t/\tau}, \tag{22}$$

where τ is a ***time constant*** that governs the rate of cooling. In general, the larger τ is (i.e., the slower the cooling), the better results one obtains; however, a larger τ also means a longer computation time, so some trial-and-error may be needed to find the optimum value for a given application.

A classic problem for which simulated annealing can provide a solution as good as or better than any other method is the famous (and famously hard-to-solve) ***traveling salesman problem***. The idea is that a salesman is to visit each of N cities once and only once, starting and ending the journey at the same city, and the goal is to minimize the total distance traveled. (One can imagine that variations on this theme might apply to GPS or airline-scheduling systems identifying the fastest

or shortest route between two or more points, or even to Internet routers attempting to find the quickest way to move data from a server to your computer.) As N grows, the computation becomes increasingly difficult: there are $N!$ possible orderings of the cities, and even for the relatively small value $N = 15$, that translates to more than 10^{12} possible orderings of the cities—more than one would want to compute the total distances for. Simulated annealing is among the best methods for solving this type of problem, but it should be emphasized that it may not always find the very best solution (the *global* minimum) for large N. However, there is no known method guaranteed of doing so without a brute force search through all the possibilities. Simulated annealing is a good choice when one wants an answer that's "good enough" and that can be found relatively quickly.

It's worth mentioning another effective optimization method inspired by the real world, which goes by the name ***genetic programming*** (based on ***genetic algorithms***). As the name might suggest, this approach adopts the biological concepts of mutation, inheritance, and natural selection, and applies them to possible solutions to an optimization problem. The idea is to start with a group of candidate solutions, evaluate their "fitness" as solutions to the problem, and then "mutate" or "mate" them to try to improve their fitness. (In the traveling salesman problem, a candidate solution would be a particular sequence of cities, and its fitness would be the total distance associated with that sequence. The solution could be mutated by, for example, interchanging two cities in the sequence. Alternatively, inheritance could be mimicked by taking two "parent" solutions and exchanging parts of their sequences of cities to produce new "child" sequences.) Genetic algorithms will not be discussed further in this set of Modules, but entire books have been written about them so the interested reader is encouraged to explore those other resources.

Recap

- Truly random numbers cannot be generated by current algorithmic computers; the best facsimiles are pseudorandom numbers generated by algorithms, some of which make use of random noise in the computer.
- Monte Carlo integration involves counting randomly placed points to determine the area inside or under a curve as a way of approximating the corresponding integral. This may require choosing the random points in a non-uniform way, in what's called importance sampling.
- Monte Carlo simulations are used to model and analyze physical systems, especially in the field of statistical mechanics. They often involve using a Markov chain approach to determining transition probabilities between states.
- Simulated annealing can be used to find the global minimum of a function; e.g., the energy function of a physical system (like the Ising model), the distance between two points chosen from among many possible routes, the "action" (from Lagrangian mechanics), and many others.

Exercise #1

(a) Write a function to use Monte Carlo integration to find the area of a disk of radius R, using the approach discussed in the Module. (b) Try your code with $R = 4.0$ and 1000 sample points, and have it print out the percent difference from the theoretical value. Do this twice, to see the variation in results due to the use of random sample points. (c) Repeat part (c) using 100000 sample points. Briefly discuss your results.

Exercise #2

(a) Run the code provided for the 2-D Ising model spanning temperatures from 0.1 to 3 (get at least 10 values in that range) and $N = 20$, and store the magnetization values. Repeat this process twice to get three values at each temperature, and then plot the *absolute values* of the three sets of magnetization values as a function of temperature on the same plot. Discuss the consistency of the three sets of results. (b) This model was found to have the following theoretical magnetization dependence:

$$M(T) = \begin{cases} 0, & \text{for } T > T_c; \\ \dfrac{(1+z^2)^{1/4}(1-6z^2+z^4)^{1/8}}{\sqrt{1-z^2}}, & \text{for } T < T_c, \end{cases} \tag{23}$$

where M is the magnitude of the magnetization (the absolute value of the sum of the lattice spins), $T_c \simeq 2.27\ J/k_B$ is the **Curie temperature**, and $z \equiv e^{-2J/k_B T}$. Plot this theoretical function together with your numerical results, and discuss their agreement (or lack thereof). What does the Curie temperature correspond to in your plot?

Exercise #3

Write a function to implement the **one**-dimensional Ising model for a lattice of N spins having periodic boundary conditions. (This means that you will want the numerical array representing the lattice, call it `latt`, to contain $N + 2$ elements, with `latt[0]` = `latt[N]` and `latt[N+1]` = `latt[1]`. The "duplicate" array elements with the `0` and `N+1` indices should be updated whenever the spins that they're copies of flip.) We want to see how the spins evolve over time, so you will want to create a 2-D array to hold the spins of the 1-D lattice at the different times (i.e., at each step in the Monte Carlo chain). With no applied magnetic field, the net magnetization of the 1-D lattice should be zero, but the lattice should exhibit **domains**—regions of spins that are aligned together— that change over time. Make a plot of your 2-D array (with the 1-D lattice along the vertical axis and "time"—the count of the steps in the Monte Carlo chain—along the horizontal axis) using the `imshow` function in `matplotlib`. In the plot, red pixels will represent $+1$ spins while blue will represent -1 (you can show the colormap using the command `colorbar` in a line following the `imshow` command). Briefly describe what your plot shows. Are there regions with spins of one sign that flip to the opposite sign? Do regions with spins of the same sign shrink or grow over time?

Breakpoint Answers

Breakpoint 1

$$\sum_i P(E_i)\,T_{ij} = \sum_i P(E_j)\,T_{ji} = P(E_j)\sum_i T_{ji} = P(E_j). \tag{24}$$

In the first step, Eq. (18) for the ratio of T's was used, and in the last step the normalization of the T_{ij}'s was employed.

Breakpoint 2

We have

$$\begin{cases} T_{ij} = \dfrac{1}{M} \times e^{-\beta(E_j - E_i)} \quad \text{and} \quad T_{ji} = \dfrac{1}{M} \times 1, & \text{if } E_j > E_i; \\ T_{ij} = \dfrac{1}{M} \times 1 \quad \text{and} \quad T_{ji} = \dfrac{1}{M} \times e^{-\beta(E_i - E_j)}, & \text{if } E_j \leq E_i. \end{cases}$$

which are supposed to obey

$$\frac{T_{ij}}{T_{ji}} = \frac{P(E_j)}{P(E_i)} = \frac{e^{-\beta E_j}/Z}{e^{-\beta E_i}/Z} = e^{-\beta(E_j - E_i)}.$$

For the first row of T values above, we compute

$$\frac{T_{ij}}{T_{ji}} = \frac{\dfrac{1}{M} \times e^{-\beta(E_j - E_i)}}{\dfrac{1}{M} \times 1} = e^{-\beta(E_j - E_i)}.$$

For the second row, we find

$$\frac{T_{ij}}{T_{ji}} = \frac{\dfrac{1}{M} \times 1}{\dfrac{1}{M} \times e^{-\beta(E_i - E_j)}} = e^{\beta(E_i - E_j)} = e^{-\beta(E_j - E_i)}.$$

So, the claim is verified.

Scientist Profile

Luis von Ahn was born in Guatemala City, Guatemala, in 1979. He is an associate professor of computer science at Cargnegie Mellon University (among the top universities for computer science in the world), having received a B.S. in mathematics from Duke University in 2000 and a Ph.D. in computer science from Carnegie Mellon in 2005. His research involves CAPTCHAs and "human computation," a term he coined in his Ph.D. dissertation for the combination of human and computer analysis used to solve problems that neither could solve alone. (CAPTCHAs are the text-based images that one is asked to interpret in order to gain access to some webpages, software downloads, etc. They are used to prevent "bots" from accessing those resources.) In 2007 he developed re-CAPTCHAs, which are based on text extracted from old books. This is a way to get human input (for free!) into the digitization of texts that cannot be analyzed by optical character recognition programs. Von Ahn has been recognized for his work with a MacArthur Fellowship ("genius grant") in 2006, a Sloan Fellowship in 2009, and many other prestigious awards. He also has been recognized by several prominent organizations and publications; for instance, he was named one of the 50 Best Brains in Science by Discover magazine, and has been listed in Popular Science magazine's "Brilliant 10." Von Ahn also has received several awards for his teaching at Carnegie Mellon. (Profile adapted from https://en.wikipedia.org/wiki/Luis_von_Ahn.)