14

Monte Carlo methods

14.1 The Monte Carlo method

The Monte Carlo method is simple, robust, and useful. It was invented by Enrico Fermi and developed by Metropolis (Metropolis et al., 1953). It has many applications. One can use it for numerical integration. One can use it to decide whether an odd signal is random noise or something to evaluate. One can use it to generate sequences of configurations that are random but occur according to a probability distribution, such as the Boltzmann distribution of statistical mechanics. One even can use it to solve virtually any problem for which one has a criterion to judge the quality of a solution and a way of generating a suitably huge space of possible solutions. That's how evolution invented us.

14.2 Numerical Integration

Suppose one wants to numerically integrate a function $f(x)$ of a vector $x = (x_1, \ldots, x_n)$ over a region $\mathcal{R}$. One generates a large number N of pseudorandom values for the n coordinates x within a hyperrectangle of length L that contains the region $\mathcal{R}$, keeps the $N_{\mathcal{R}}$ points $x_k = (x_{1k}, \ldots, x_{nk})$ that fall within the region $\mathcal{R}$, computes the average $\langle f(x_k) \rangle$, and multiplies by the hypervolume $V_{\mathcal{R}}$ of the region

$$\int_{\mathcal{R}} f(x)\, d^n x \approx \frac{V_{\mathcal{R}}}{N_{\mathcal{R}}} \sum_{k=1}^{N_{\mathcal{R}}} f(x_k). \tag{14.1}$$

If the hypervolume $V_{\mathcal{R}}$ is hard to compute, you can have the Monte Carlo code compute it for you. The hypervolume $V_{\mathcal{R}}$ is the volume L^n of the enclosing hypercube multiplied by the number $N_{\mathcal{R}}$ of times the N points

fall within the region $\mathcal{R}$

$$V_\mathcal{R} = \frac{N_\mathcal{R}}{N} L^n. \tag{14.2}$$

The integral formula (14.1) then becomes

$$\int_\mathcal{R} f(x) \, d^n x \approx \frac{L^n}{N} \sum_{k=1}^{N_\mathcal{R}} f(x_k). \tag{14.3}$$

The error falls like $1/\sqrt{N_\mathcal{R}}$. So to get better accuracy, one simply runs the program again with more points; one does not have to write a new code with finer n-dimensional grids of points that span the region $\mathcal{R}$.

Example 14.1 (Numerical Integration). Suppose one wants to integrate the function

$$f(x,y) = \frac{e^{-2x-3y}}{\sqrt{x^2 + y^2 + 1}} \tag{14.4}$$

over the quarter of the unit disk in which x and y are positive. In this case, $V_\mathcal{R}$ is the area $\pi/4$ of the quarter disk.

To generate fresh random numbers, one must set the seed for the code that computes them. The following program sets the seed by using the subroutine init_random_seed defined in a FORTRAN95 program in section 13.20. With some compilers, one can just write "call random_seed()."

```
program integrate
  implicit none ! catches typos
  integer :: k, N
  integer, parameter :: dp = kind(1.0d0)
  real(dp) :: x, y, sum = 0.0d0, f
  real(dp), dimension(2) :: rdn
  real(dp), parameter :: area = atan(1.0d0) ! pi/4
  f(x,y) = exp(-2*x - 3*y)/sqrt(x**2 + y**2 + 1.0d0)
  write(6,*) 'How many points?'
  read(5,*) N
  call init_random_seed() ! set new seed
  do k = 1, N
10    call random_number(rdn); x= rdn(1); y = rdn(2)
      if (x**2+y**2 > 1.0d0) then
         go to 10
      end if
      sum = sum + f(x,y)
  end do
```

```
  ! integral = area times mean value < f > of f
  sum = area*sum/real(N,dp)
  write(6,*) 'The integral is ',sum
end program integrate
```

I ran this code with npoints $= 10^\ell$ for $\ell = 1, 2, 3, 4, 5, 6, 7$, and 8 and found respectively the results 0.059285, 0.113487, 0.119062, 0.115573, 0.118349, 0.117862, 0.117868, and 0.117898. The integral is approximately 0.1179.

An equivalent C++ code by Sean Cahill is:

```cpp
#include <math.h>
#include <iostream>
#include <stdlib.h>

using namespace std;

// The function to integrate
double f(const double& x, const double& y)
{
  double numer = exp(-2*x - 3*y);
  double denom = sqrt(x*x + y*y + 1);
  double retval = numer / denom;

  return retval;
}

void integrate ()
{
  // Declares local constants
  const double area = atan(1); // pi/4

  // Inits local variables
  int n=0;
  double sum=0,x=0,y=0;

  // Seeds random number generator
  srand ( time(NULL) );

  // Gets the value of N
  cout << "What is N? ";
```

```
  cin >> n;

  // Loops the given number of times
  for (int i=0; i<n; i++)
  {
    // Loops until criteria met
    while (true)
    {
      // Generates random points between 0 and 1
      x = static_cast<double>(rand()) / RAND_MAX;
      y = static_cast<double>(rand()) / RAND_MAX;

      // Checks if the points are suitable
      if ((x*x + y*y) <= 1)
      {
        // If so, break out of the while loop
        break;
      }
    }

    // Updates our sum with the given points
    sum += f(x,y);
  }

  // Integral = area times mean value < f > of f
  sum = area * sum / n;

  cout << "The integral is " << sum << endl;
}
```

14.3 Quasirandom numbers

The method of the previous section is easy to use, but one can improve its accuracy by using **quasirandom numbers**, which occur with equal density in every region. Examples on the interval $(0, 1)$ are the **Halton** sequences of bases 2 and 3

$$1/2,\ 1/4,\ 3/4,\ 1/8,\ 5/8,\ 7/8,\ 1/16,\ 9/16, \ldots$$
$$1/3,\ 2/3,\ 1/9,\ 4/9,\ 7/9,\ 2/9,\ 5/9,\ 8/9,\ 1/27, \ldots \tag{14.5}$$

and those invented by Sobol'. If one uses quasirandom numbers instead of pseudorandom numbers to estimate integrals, then the error falls like $1/N^{2/3}$ or $1/N$, both of which are faster than the $1/\sqrt{N}$ decrease one gets with pseudorandom numbers. Codes that generate Sobol' sequences are available in C (Press et al., 2007, sec. 7.8) and in Fortran (Press et al., 1997, sec. 7.7).

14.4 Applications to Experiments

Physicists accumulate vast quantities of data and sometimes must decide whether a particular signal is due to a defect in the detector, to a random fluctuation in the real events that they are measuring, or to a new and unexpected phenomenon. For simplicity, let us assume that the background can be ignored and that the real events arrive randomly in time apart from extraordinary phenomena. One reliable way to evaluate an ambiguous signal is to run a Monte Carlo program that generates the kinds of real random events to which one's detector is sensitive and to use these events to compute the probability that the unusual signal occurred randomly.

To illustrate the use of random-event generators, we will consider the work of a graduate student who spent 100 days counting muons produced in an underground detector by atmospheric GeV neutrinos. Each of the very large number N of primary cosmic rays that hit the Earth every day can collide with a nucleus and make a shower of pions which in turn produce atmospheric neutrinos that can make muons in the detector. The probability p that a given cosmic ray will make a muon in the detector is very small, but the number N of primary cosmic rays is very large. In this experiment, their product pN was $\langle n \rangle = 0.1$ muons per day. Since N is huge and p tiny, the probability distribution is Poisson, and so by (13.58) the probability that n muons would be detected on any particular day is

$$P(n, \langle n \rangle) = \frac{\langle n \rangle^n}{n!}\, e^{-\langle n \rangle} \tag{14.6}$$

in the absence of a failure of the anti-coincidence shield or some other problem with the detector—or some hard-to-imagine astrophysical event.

The graduate student might have used the following program to generate 1,000,000 random histories of 100 days of events distributed according to the Poisson distribution (14.6) with $\langle n \rangle = 0.1$:

```
program muons
  implicit none
```

```fortran
   integer(kind=8) :: k, m, day, number
   integer(kind=8), parameter :: N = 1000000 ! number of data sets
   integer(kind=8), dimension(N,100) :: histories
   integer(kind=8), dimension(0:100) :: maxEvents = 0, sumEvents = 0
   real(kind=8) :: prob, x, numMuons, totMuons
   real(kind=8), dimension(0:100) :: p
   real(kind=8), parameter :: an = 0.1 ! <n> events per day
   prob = exp(-an); p(0) = prob; maxEvents = 0
   ! p(k) is the probability of fewer than k+1 events per day
   do k = 1, 100 ! make Poisson distribution
      prob = prob + an**k*exp(-an)/gamma(real(k+1,kind=8))
      p(k) = prob
   end do
   call init_random_seed() ! sets random seed
   do k = 1, N ! do N histories
      do day = 1, 100 ! do day of kth history
         call random_number(x)
         do m = 100, 0, -1
            if (x < p(m)) then
               number = m
            end if
         end do
         histories(k,day) = number
      end do
      numMuons = maxval(histories(k,:))
      totMuons = sum(histories(k,:))
      maxEvents(numMuons) = maxEvents(numMuons) + 1
      sumEvents(totMuons) = sumEvents(totMuons) + 1
   end do
   open(7,file="maxEvents"); open(8,file="totEvents")
   do k = 0, 100
      write(7,*) k, maxEvents(k); write(8,*) k, sumEvents(k)
   end do
end program muons
```

Figure 14.1 plots the results from this simple Monte Carlo of 1,000,000 runs or histories of 100 days each. The boxes show that the maximum number of muons detected on a single day respectively was $n = 1, 2$, and 3 on 62.6%, 35.9%, and 1.5% of the runs—and respectively was $n = 0, 4, 5$, and 6 on only 36, 410, 9, and 1 runs. Thus if the actual run detected no muons at
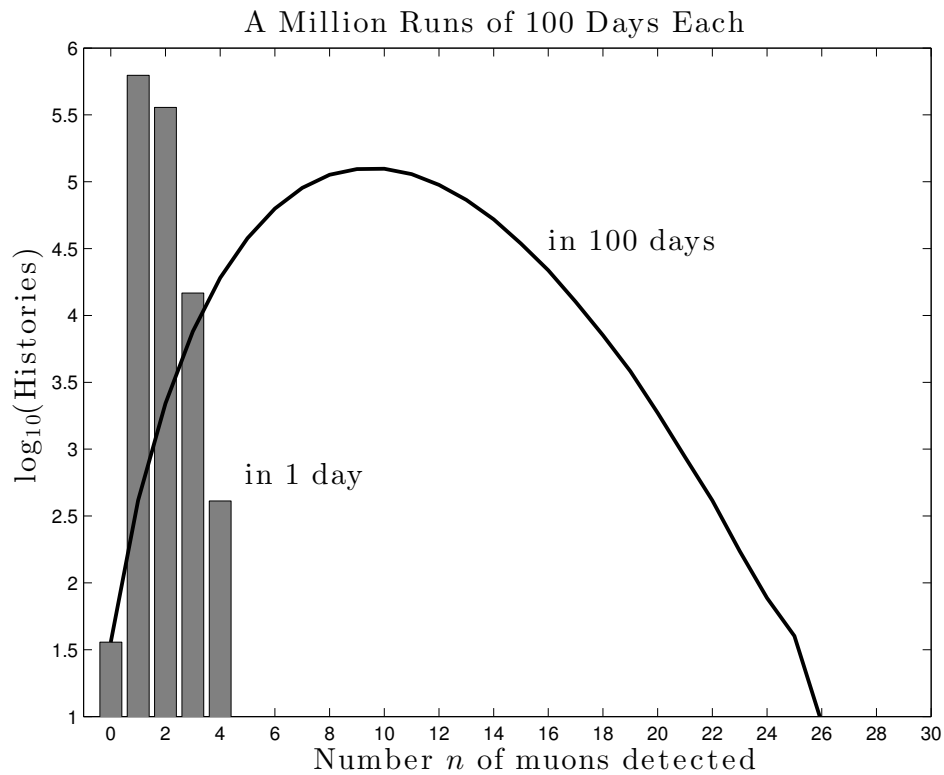
A Million Runs of 100 Days Each

Figure 14.1 The number (out of 1,000,000) of histories of 100 days in which a maximum of n muons is detected on a single day (boxes) and in 100 days (curve).

all, that would be by (13.83) about a 4σ event, while a run with more than 4 muons on a single day would be an event of more than 4σ. Either would be a reason to examine the apparatus or the heavens; the Monte Carlo can't tell us which. The curve shows how many runs had a total of n muons in 100 days; 125,142 histories had 10 muons.

Of course, one could compute the data of Fig. 14.1 by hand without running a Monte Carlo. But suppose one's aging phototubes reduced the mean number of muons detected per day to $\langle n \rangle = 0.1(1 - \alpha d/100)$ on day d? Or suppose one needed the probability of detecting more than one muon on two days separated by one day of zero muons? In such cases, the analytic computation would be difficult and error prone, but the student would need to change only a few lines in the Monte Carlo program.

An equivalent C++ code by Sean Cahill is:

```
#include <stdlib.h>
```

```cpp
#include <time.h>
#include <math.h>
#include <iostream>
#include <fstream>
#include <iomanip>
#include <vector>
#include <valarray>

using namespace std;

// Calculates the factorial of n
double factorial(const int& n)
{
  double f = 1;

  int i=0;
  for(i = 1; i <= n; i++)
  {
    f *= i;
  }
  return f;
}

void muons()
{
  // Declares constants
  const int N = 1000000; // Number of data sets
  const int LOOP_ITR = 101;
  const double AN = 0.1; // Number of events per day

  // Inits local variables
  int k=0, m=0, day=0, num=0, numMuons=0, totMuons=0;
  int maxEvents[LOOP_ITR];
  int totEvents[LOOP_ITR];
  memset (maxEvents, 0, sizeof(int) * LOOP_ITR);
  memset (totEvents, 0, sizeof(int) * LOOP_ITR);

  // Creates our 2d histories array
  vector<valarray<int> > histories(N, LOOP_ITR);
```

```
double prob=0,tmpProb=0,fact=0, x=0;
double p[LOOP_ITR];

// probability of no events
p[0] = exp(-AN);
prob = p[0];

// p(k) is the probability of fewer than k+1 events per day
for (k=1; k<=LOOP_ITR; k++)
{
  fact = factorial (k);
  tmpProb = k * exp(-AN) / fact;
  prob += pow(AN, tmpProb);
  p[k] = prob;
}

// Random seed
srand ( time(NULL) );

// Goes through all the histories
for (k=0; k<N; k++)
{
  // Goes through all the days
  for (day=1; day<LOOP_ITR; day++)
  {
    // Generates a random number betwen 0 and 1
    x = static_cast<double>(rand()) / RAND_MAX;

    // Finds an M with p(M) < X
    for (m=100; m>=0; m--)
    {
      if (x < p[m])
      {
        num = m;
      }
    }

    histories[k][day] = num;
  }
```

```
    // Calculates max and sum
    numMuons = histories[k].max();
    totMuons = histories[k].sum();

    // Updates our records
    maxEvents[numMuons]++;
    totEvents[totMuons]++;
  }

  // Opens a data file
  ofstream fhMaxEvents, fhSumEvents;
  fhMaxEvents.open ("maxEvents.txt");
  fhSumEvents.open ("totEvents.txt");

  // Sets precision
  fhMaxEvents.setf(ios::fixed,ios::floatfield);
  fhMaxEvents.precision(7);
  fhSumEvents.setf(ios::fixed,ios::floatfield);
  fhSumEvents.precision(7);

  // Writes the data to a file
  for (k=0; k<LOOP_ITR; k++)
  {
    fhMaxEvents << k << "   " << maxEvents[k] << endl;
    fhSumEvents << k << "   " << totEvents[k] << endl;
  }
}
```

14.5 Statistical Mechanics

The Metropolis algorithm can generate a sequence of states or configurations of a system distributed according to the Boltzmann probability distribution (1.351). Suppose the state of the system is described by a vector x of many components. For instance, if the system is a protein, the vector x might be the $3N$ spatial coordinates of the N atoms of the protein. A protein composed of 200 amino acids has about 4000 atoms, and so the vector x would have some 12,000 components. Suppose $E(x)$ is the energy of configuration

x of the protein in its cellular environment of salty water crowded with macromolecules. How do we generate a sequence of "native states" of the protein at temperature T?

We start with some random or artificial initial configuration x^0 and then make random changes δx in successive configurations x. One way to do this is to make a small, random change δx_i in coordinate x_i and then to test whether to accept this change by comparing the energies $E(x)$ and $E(x')$ of the two configurations x and x', which differ by δx_i in coordinate x_i. (Estimating these energies is not trivial; Gromacs and TINKER can help.) It is important that this random walk be symmetric, that is, the choice of testing whether to go from x to x' when one is at x should be exactly as likely as the choice of testing whether to go from x' to x when one is at x'. Also, the sequences of configurations should be **ergodic**; that is, from any configuration x, one should be able to get to any other configuration x' by a suitable sequence of changes $\delta x_i = x'_i - x_i$.

How do we decide whether to accept or reject δx_i? We use the following **Metropolis step**: If the energy $E' = E(x')$ of the new configuration x' is less than the energy $E(x)$ of the current configuration x, then we accept the new configuration x'. If $E' > E$, then we accept x' with probability

$$P(x \to x') = e^{-(E'-E)/kT}. \tag{14.7}$$

In practice, one generates a pseudorandom number $r \in [0, 1]$ and accepts x' if

$$r < e^{-(E'-E)/kT}. \tag{14.8}$$

If one does not accept x', then the system remains in configuration x.

In FORTRAN90, the Metropolis step might be

```
if ( newE <= oldE ) then ! accept
   x(i) = x(i) + dx
else ! accept conditionally
   call random_number(r)
   if ( r <= exp(-beta*(newE - oldE)) ) then ! accept
      x(i) = x(i) + dx
   end if
end if
```

in which $\beta = 1/kT$.

One then varies another coordinate, such as x_{i+1}. Once one has varied all of the coordinates, one has finished a **sweep** through the system. After thousands or millions of such sweeps, the protein is said to be **thermalized**.

Once the system is thermalized, one can start measuring properties of the system. One computes a physical quantity every hundred or every thousand sweeps and takes the average of these measurements. That average is the mean value of the physical quantity at temperature T.

Why does this work? Consider two configurations x and x' which respectively have energies $E = E(x)$ and $E' = E(x')$ and are occupied with probabilities $P_t(x)$ and $P_t(x')$ as the system is thermalizing. If $E > E'$, then the rate $R(x \to x')$ of going from x to x' is the rate v of choosing to test x' when one is at x times the probability $P_t(x)$ of being at x, that is, $R(x \to x') = v\,P_t(x)$. The reverse rate $R(x' \to x)$ is $R(x' \to x) = v\,P_t(x')\,e^{-(E-E')/kT}$ with the same v since the random walk is symmetric. The net rate from $x \to x'$ then is

$$R(x \to x') - R(x' \to x) = v\left(P_t(x) - P_t(x')\,e^{-(E-E')/kT}\right). \qquad (14.9)$$

This net flow of probability from $x' \to x$ is positive if and only if

$$P_t(x)/P_t(x') > e^{-(E-E')/kT}. \qquad (14.10)$$

The probability distribution $P_t(x)$ therefore flows with each sweep toward the Boltzmann distribution $\exp(-E(x)/kT)$. The flow slows and stops when the two rates are equal $R(x' \to x) = R(x \to x')$ a condition called **detailed balance**. At this equilibrium, the distribution $P_t(x)$ satisfies

$$P_t(x) = P_t(x')\,e^{-(E-E')/kT} \qquad (14.11)$$

in which $P_t(x')\,e^{E'/kT}$ is independent of x. So the thermalizing distribution $P_t(x)$ approaches the distribution $P(x) = c\,e^{-E/kT}$ in which c is independent of x. Since the sum of these probabilities must be unity, we have

$$\sum_x P(x) = c\sum_x e^{-E(x)/kT} = 1 \qquad (14.12)$$

which means that the constant c is the inverse of the **partition function**

$$Z(T) = \sum_x e^{-E(x)/kT}. \qquad (14.13)$$

The thermalizing distribution approaches Boltzmann's distribution (1.351)

$$P_t(x) \to P_B(x) = e^{-E(x)/kT}/Z(T). \qquad (14.14)$$

Example 14.2 (Z_2 Lattice Gauge Theory). First, one replaces spacetime with a lattice of points in d dimensions. Two nearest neighbor points are separated by the lattice spacing a and joined by a link. Next, one puts an

element U of the gauge group on each link. For the Z_2 gauge group (example 10.4), one assigns an action $S_\square$ to each elementary square or *plaquette* of the lattice with vertices 1, 2, 3, and 4

$$S_\square = 1 - U_{1,2}\, U_{2,3}\, U_{3,4}\, U_{4,1}. \tag{14.15}$$

Then, one replaces $E(x)/kT$ with βS in which the action S is a sum of all the plaquette actions S_p. More details are available at Michael Creutz's website (latticeguy.net/lattice.html). $\square$

Example 14.3 ($SU(3)$ lattice gauge theory)**.** For each elementary square of the lattice, the plaquette variable U_p is the product of elements U of the gauge group $SU(3)$ around the square, $U_p = U_{1,2}\, U_{2,3}\, U_{3,4}\, U_{4,1}$. The euclidian action of the theory is then the sum over all the plaquettes of the lattice of the traces

$$S = \beta \sum_p \left[1 - \frac{1}{6}\mathrm{Tr}\left(U_p + U_p^\dagger \right) \right] \tag{14.16}$$

in which $\beta = 6/g^2$ is inversely proportional to the coupling constant g. $\square$

Although the generation of configurations distributed according to the Boltzmann probability distribution (1.351) is one of its most useful applications, the Monte Carlo method is much more general. It can generate configurations x distributed according to any probability distribution $P(x)$.

To generate configurations distributed according to $P(x)$, we accept any new configuration x' if $P(x') > P(x)$ and also accept x' with probability

$$P(x \to x') = P(x')/P(x) \tag{14.17}$$

if $P(x) > P(x')$.

This works for the same reason that the Boltzmann version works. Consider two configurations x and x'. After the system has thermalized, the probabilities $P_t(x)$ and $P_t(x')$ have reached equilibrium, and so the rate $R(x \to x')$ from $x \to x'$ must equal the rate $R(x' \to x)$ from $x' \to x$. If $P(x') > P(x)$, then $R(x \to x')$ is

$$R(x \to x') = v\, P_t(x) \tag{14.18}$$

in which v is the rate of choosing $\delta x = x' - x$, while the rate $R(x' \to x)$ is

$$R(x' \to x) = v\, P_t(x')\, P(x)/P(x') \tag{14.19}$$

with the same v since the random walk is symmetric. Equating the two rates $R(x \to x') = R(x' \to x)$, we find that after thermalization

$$P_t(x) = P(x)\, P_t(x')/P(x') = c\, P(x) \tag{14.20}$$

in which c is independent of x. Thus $P_t(x)$ converges to $P(x)$ at equilibrium.

So far we have assumed that the rate of choosing $x \to x'$ is the same as the rate of choosing $x' \to x$. In **Smart Monte Carlo** schemes, physicists arrange the rates $v_{x \to x'}$ and $v_{x' \to x}$ so as to steer the flow and speed-up thermalization. To compensate for this asymmetry, they change the second part of the Metropolis step from $x \to x'$ when $P(x) > P(x')$ to accept conditionally with probability

$$P(x \to x') = P(x')\, v_{x' \to x} / \left[P(x)\, v_{x \to x'} \right]. \tag{14.21}$$

Now if $P(x) > P(x')$, then $R(x' \to x)$ is

$$R(x' \to x) = v_{x' \to x}\, P_t(x') \tag{14.22}$$

while the rate $R(x \to x')$ is

$$R(x \to x') = v_{x \to x'}\, P_t(x)\, P(x')\, v_{x' \to x} / \left[P(x)\, v_{x \to x'} \right]. \tag{14.23}$$

Equating the two rates $R(x' \to x) = R(x \to x')$, we find

$$P_t(x) = P(x)\, P_t(x') / P(x') \tag{14.24}$$

which implies that $P_t(x)$ converges to

$$P_t(x) = P(x)/Z \tag{14.25}$$

in which $Z = \int P(x)\, dx$.

Example 14.4 (Highly multiple integration)**.** You can use the general Metropolis method (14.17–14.20) to integrate a function $f(x)$ of many variables $x = (x_1, \dots, x_n)$ if you can find a positive function $g(x)$ similar to $f(x)$ whose integral $I[g]$ you know. You just use the probability distribution $P(x) = g(x)/I[g]$ to find the mean value of $I[g] f(x)/g(x)$:

$$\int f(x)\, d^n x = I[g] \int \frac{f(x)}{g(x)} \frac{g(x)}{I[g]}\, d^n x = I[g] \int \frac{f(x)}{g(x)} P(x)\, d^n x. \tag{14.26}$$

$\square$

14.6 Solving Arbitrary Problems

If you know how to generate a suitably large space of trial solutions to a problem, and you also know how to compare the quality of any two of your solutions, then you can use a Monte Carlo method to solve the problem. The hard parts of this seemingly magical method are characterizing a big enough

space of solutions s and constructing a quality function or functional that assigns a number $Q(s)$ to every solution in such a way that if s is a better solution than s', then

$$Q(s) > Q(s'). \tag{14.27}$$

But once one has characterized the space of possible solutions s and has constructed the quality function $Q(s)$, then one simply generates zillions of random solutions and selects the one that maximizes the function $Q(s)$ over the space of all solutions.

If one can characterize the solutions as vectors of a certain dimension, $s = (x_1, \ldots, x_n)$, then one may use the Monte Carlo method of the previous section (14.5) by setting $P(s) = Q(s)$.

14.7 Evolution

The reader may think that the use of Monte Carlo methods to solve arbitrary problems is quite a stretch. Yet Nature has applied them to the problem of evolving species that survive. As a measure of the quality $Q(s)$ of a given solution s, Nature used the time derivative of the logarithm of its population $\dot{P}(t)/P(t)$. The space of solutions is the set of possible genomes. Leaving aside DNA methylation, histone modification, and other epigenetic changes, we may idealize each solution or genome as a sequence of nucleotides $s = b_1 b_2 \ldots b_N$ some thousands or billions of bases long, each base b_k being adenine, cytosine, guanine, or thymine (A, C, G, or T). Since there are four choices for each base, the set of solutions is huge. The genome for *homo sapiens* has some 3 billion bases (or base pairs, DNA being double stranded), and so the solution space is a set with

$$\mathcal{N} = 4^{3 \times 10^9} = 10^{1.8 \times 10^9} \tag{14.28}$$

elements. By comparison, a google is only 10^{100}.

In evolution, a Metropolis step begins with a random change in the sequence of bases; changes in a germ-line cell can create a new individual. Some of these changes are due to errors in the normal mechanisms by which genomes are copied and repaired. The (holo)enzyme DNA polymerase copies DNA with remarkable fidelity, making one error in every billion base pairs copied. Along a given line of descent, only about one nucleotide pair in a thousand is randomly changed in the germ line every million years. Yet in a population of 10,000 diploid individuals, every possible nucleotide substitution will have been tried out on about 20 occasions during a million years (Alberts et al., 2008).

RNA polymerases transcribe DNA into RNA, and RNAs play many roles: Ribosomes translate messenger RNAs (mRNAs) into proteins, which are sequences of amino acids; ribosomal RNAs (rRNAs) combine with proteins to form ribosomes; long noncoding RNAs (ncRNAs) regulate the rates at which different genes are transcribed; micro RNAs (miRNAs) regulate the rates at which different mRNAs are translated into proteins; and other RNAs have other as yet unknown functions. So a change of one base, *e.g.* from A to C, might alter a protein or change the expression of a gene or be silent.

Sexual reproduction makes bigger random changes in genomes. In **meiosis**, the paternal and maternal versions of each of our 23 chromosomes are duplicated, and the four versions swap segments of DNA in a process called genetic recombination or crossing-over. The cell then divides twice producing four **haploid** germ cells each with a single paternal, maternal, or mixed version of each chromosome. Two haploid cells, one from each parent, join to start a new individual. This second kind of Metropolis step makes evolution more ergodic, which is why most complex modern organisms use sexual reproduction.

Other genomic changes occur when a virus inserts its DNA into that of a cell and when transposable elements (transposons) of DNA move to different sites in a genome.

In evolution, the rest of the Metropolis step is done by the new individual: if he or she survives and multiplies, then the change is accepted; if he or she dies without progeny, then the change is rejected. Evolution is slow, but it has succeeded in turning a soup of simple molecules into humans with brains of 100 billion neurons, each with 1000 connections to other neurons.

John Holland and others have incorporated analogs of these Metropolis steps into Monte Carlo techniques called **genetic algorithms** for solving wide classes of problems (Holland, 1975; Vose, 1999; Schmitt, 2001).

Evolution also occurs at the cellular level when a cell mutates enough to escape the control imposed on its proliferation by its neighbors and transforms into a cancer cell.

Further Reading

The classic *Quarks, Gluons, and Lattices* (Creutz, 1983) is a marvelous introduction to the subject; his website (latticeguy.net/lattice.html) is an extraordinary resource, as is Rubinstein's *Simulation and the Monte Carlo Method* (Rubinstein and Kroese, 2007).

Exercises

14.1 Go to Michael Creutz's website (latticeguy.net/lattice.html) and get his C-code for Z_2 lattice gauge theory. Compile and run it, and make a graph that exhibits strong hysteresis as you raise and lower $\beta = 1/kT$.

14.2 Modify his code and produce a graph showing the coexistence of two phases at the critical coupling $\beta_t = 0.5 \ln(1 + \sqrt{2})$. Hint: Do a cold start and then 100 updates at β_t, then do a random start and do 100 updates at β_t. Plot the values of the action against the update number $1, 2, 3, \ldots 100$.

14.3 Modify Creutz's C code for Z_2 lattice gauge theory so as to be able to vary the dimension d of spacetime. Show that for $d = 2$, there's no hysteresis loop (there's no phase transition). For $d = 3$, show that any hysteresis loop is minimal (there's a second-order phase transition).

14.4 What happens when $d = 5$?

14.5 Use example 14.4 to compute the ten-dimensional integral

$$\mathcal{I} = \int \exp\left[-\left(x^2 + (x^2)^2\right)\right] d^{10}x \tag{14.29}$$

over $\mathbb{R}^{10}$ where $x^2 = x_1^2 + \ldots + x_{10}^2$.