AEP 4380 HW#9: Monte Carlo Calculations

Gianfranco Grillo

November 23, 2016

1 Background

1.1 Random Number Generator and Monte Carlo

Certain problems that arise in the physical sciences are based on processes whose complexity is too overwhelming for humans to be able to tackle them analytically. Although the underlying principles are deterministic, they may combine in ways that result in behavior that is far too complex for us to be able to perfectly model. Many such problems are instead tackled via the use of random number generators. The authors of *Numerical Recipes* point out that it seems counterintuitive that we are able to use completely deterministic machines like computers in order to generate sequences of random numbers, and yet relatively simple algorithms work as RNG without causing the universe to implode. It seems to me that this intuition is misguided: it is generally agreed that the macroscopic laws of nature are also deterministic, and yet we constantly observe randomness in nature, even in the case of simple processes. Thus, it doesn't seem surprising that computers can produce random output.

Numerical algorithms that rely on some kind of random data in order to solve problems in the sciences are known as Monte Carlo methods. They are usually used as a last resort, in order to attempt to numerically solve problems that are so intractable that we are not capable of solving via any other means. Monte Carlo simulations are essential in field like biophysics, where we are often confronted with problems containing multiple coupled degrees of freedom that are far to complex to model analytically. This homework deals with one such problem, protein folding, the process in which proteins are assembled on the basis of long chains of aminoacids.

1.2 Protein Folding

Proteins are an essential component of living organisms. They are a type of macromolecule that can be modeled as series of interconnected amino acids. The particular way in which the sequence of amino acids is arranged, as well as the types of amino acids involved, determine the specific characteristics that the protein will have. The process through which proteins are assembled, starting from an amino acid chain, is known as protein folding. This process can be modeled via a Monte Carlo simulation. Assuming the proteins are confined to Edward Abbott's world of Flatland, that is, they are two dimensional (although they can use the third dimension to fold themselvelves up), we can make use of random procedures to look at how the protein will fold itself over time, by making sure the system follows certain rules related to its total energy and the way different configurations relate to this total energy. Amino acids on the chain that are close to each other experience Van der Waals forces. This implies that the system has a certain amount of energy associated to it. This energy E_{total} is given by

$$E_{total} = \sum_{all\,pairs} E_{t(i),t(j)}\delta_{ij} = \frac{1}{2}\sum_{j\neq i} E_{t(i),t(j)}\delta_{ij} \tag{1}$$

where $t(i)$ and $t(j)$ are two different amino acids, and δ_{ij} is the Kronecker delta. We have ignored the energies bonding each of the amino acids together since it is assumed that this energy stays constant (that linked amino acids are always separated by the same distance). Thus the energies in (1) arise from Van der Waals forces between amino acids that are close to each other (more specifically, that are separated by one square in our model's two dimensional grid) and not from the forces linking the amino acids together. The value of each interaction energy

$E_{t(i),t(j)}$ is determined randomly at the start from a preset range, and it changes depending on the types of amino acids involved, which are also determined at the start of each simulation. At each time step, an amino acid is picked at random and attempts to fold in a random direction that keeps the integrity of the chain intact. If the energy of the new configuration is less or equal than the energy of the configuration at the beginning of the time step, then the folding step is performed. If it is not, then the step has a probability of being accepted $p(\Delta E)$ given by

$$p(\Delta E) = \exp\left(\frac{-\Delta E}{k_B T}\right) \tag{2}$$

where k_B is the Boltzmann constant, T is the temperature of the environment, and $\Delta E = E_{new} - E_{old}$. This procedure can be repeated any number of times. In this homework, the allowed range for the energy interactions was between -7 and -2 (in arbitrary units), 20 different types of amino acids were used, $k_B T$ is set equal to 1, and the procedure was performed over 10 million time steps, starting with a horizontally straight chain of 45 amino acids.

2 Results

2.1 Task 1

Task 1 consisted in testing the RNG that was used later in the simulation. Figure 1 shows an histogram of 100,000 random numbers generated by the generator in the range between 0 and 1. Figure 2 shows a plot of 10,000 pairs of random numbers generated in the same range, where each pair of numbers corresponded to an x and y coordinate. The code used to generate the data used to produce the plots is shows in section 4.1. The RNG used is from [1].

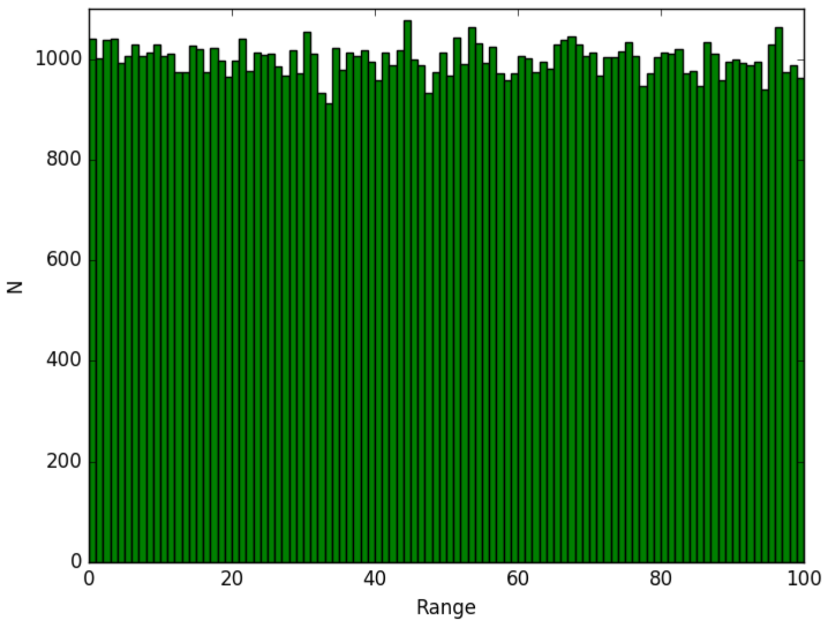

Figure 1: Histogram of 100,000 randomly generated numbers

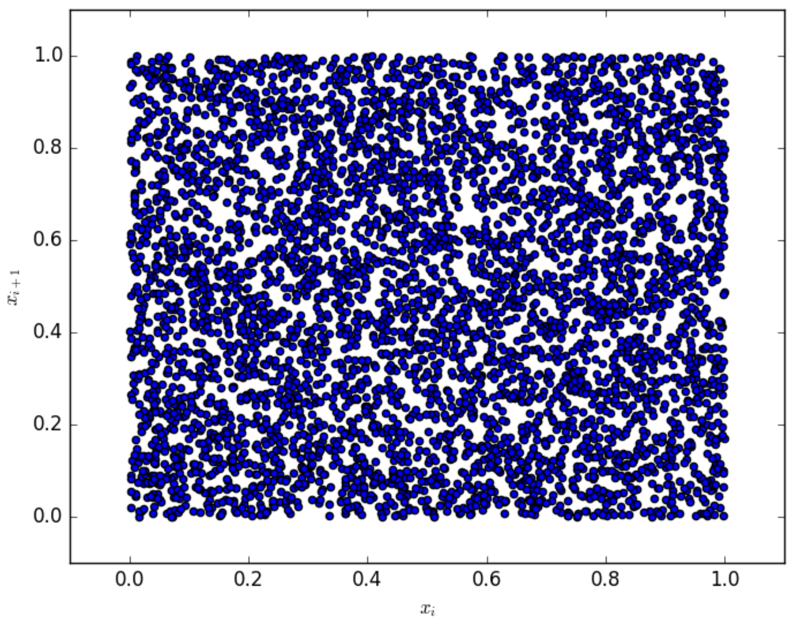

Figure 2: Plot of 10,000 randomly generated number pairs

2.2 Task 2

The second task involved performing the simulation described in section 1.2 and plotting the end-to-end distance as a function of time step, as well as the system energy as a function of time step, which are shown in Figure 3 and Figure 4, respectively. Actual plots of the amino acid chain shape are shown in Figures 5-8 at time steps $t_1 = 10^4$, $t_2 = 10^5$, $t_3 = 10^6$, and $t_4 = 10^7$, respectively. The code used is given in 4.2.

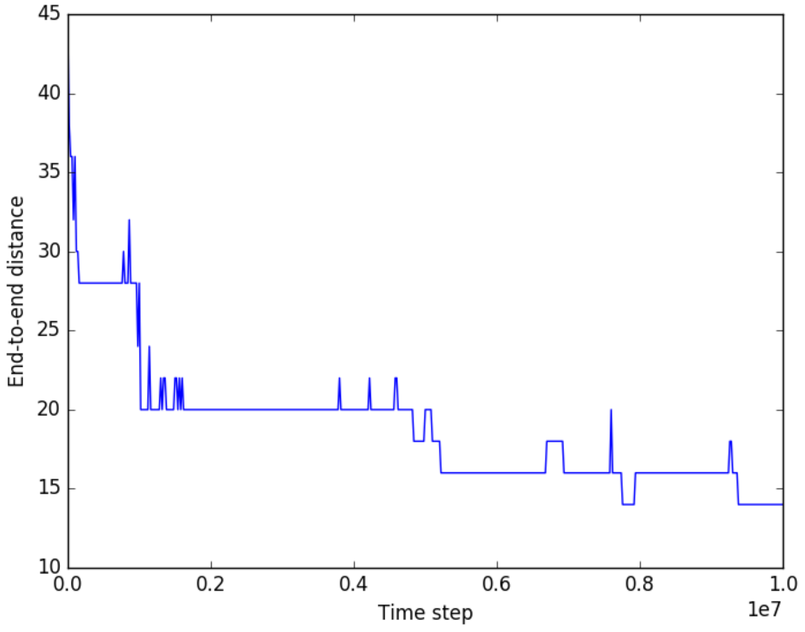

Figure 3: End-to-end chain distance vs time step

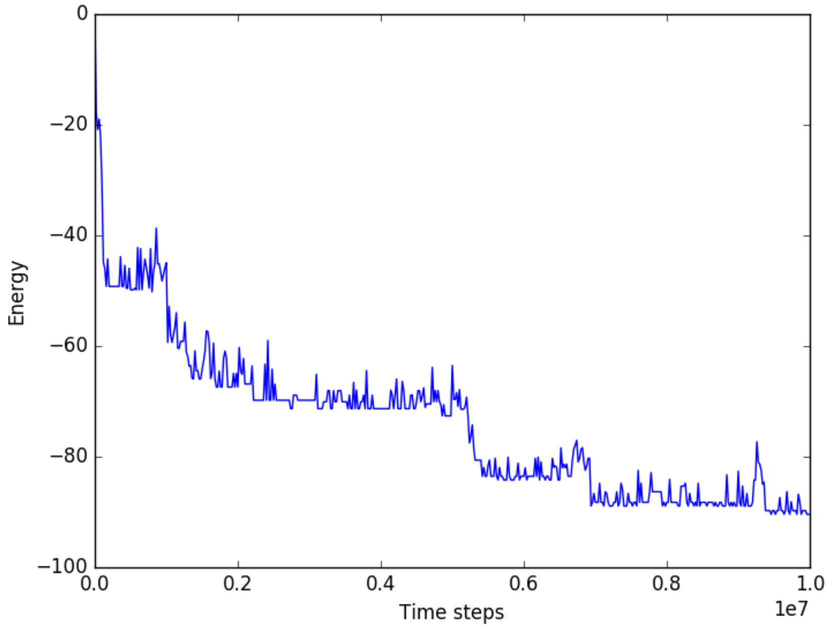

Figure 4: System energy vs time step

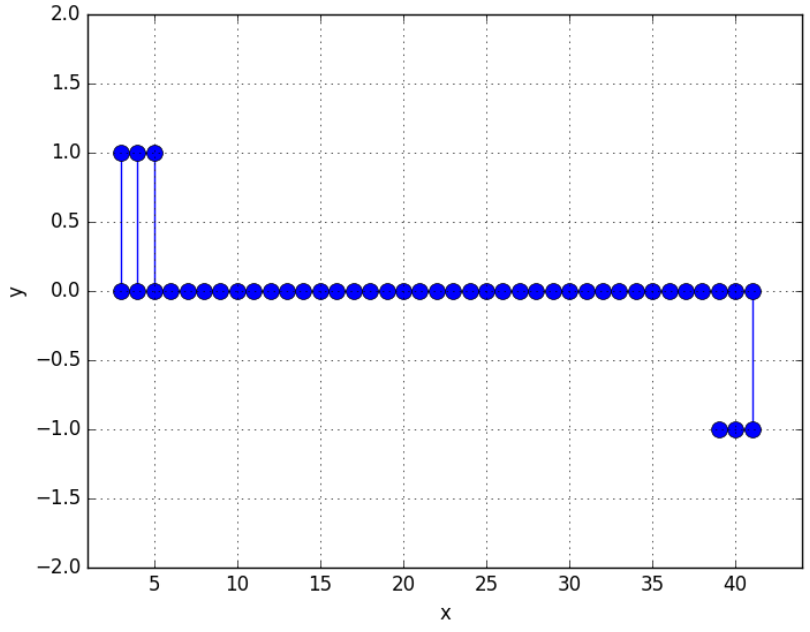

Figure 5: Amino acid chain shape at $t_1 = 10^4$

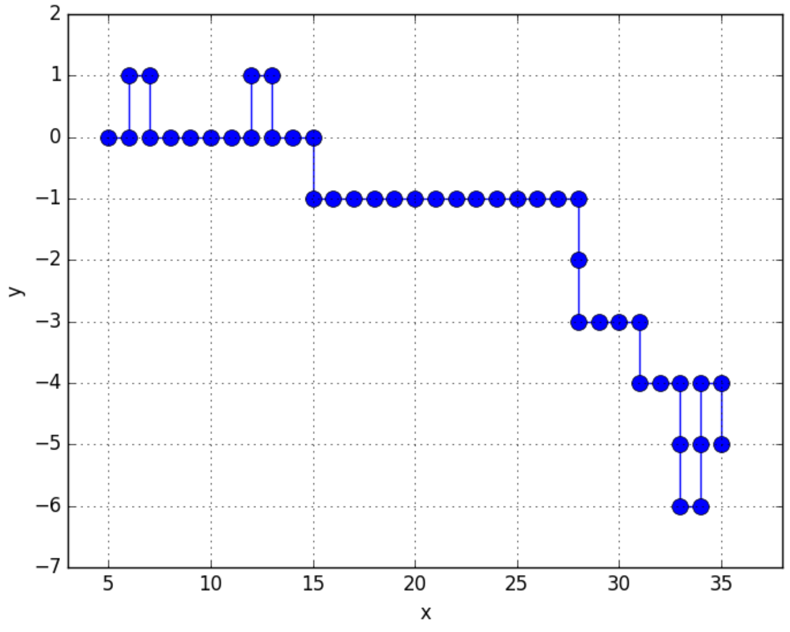

Figure 6: Amino acid chain shape at $t_2 = 10^5$

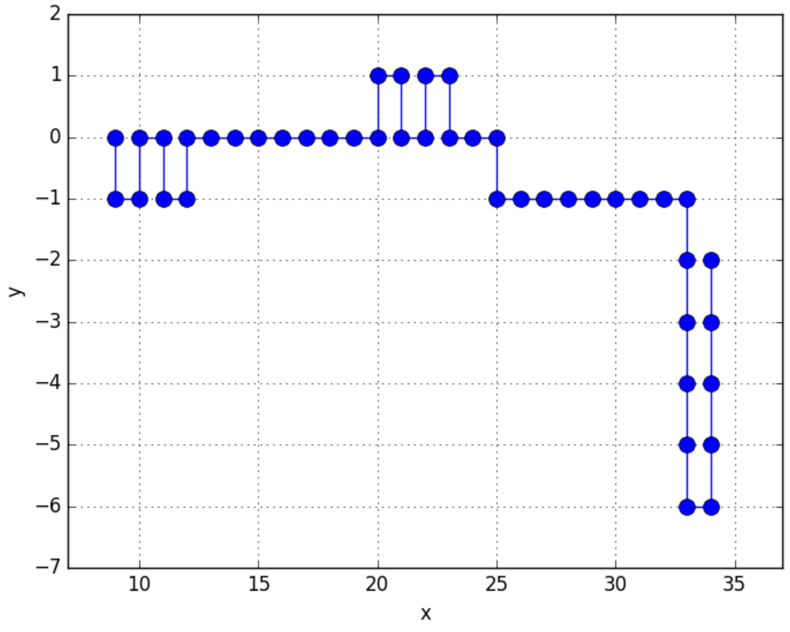

Figure 7: Amino acid chain shape at $t_3 = 10^6$

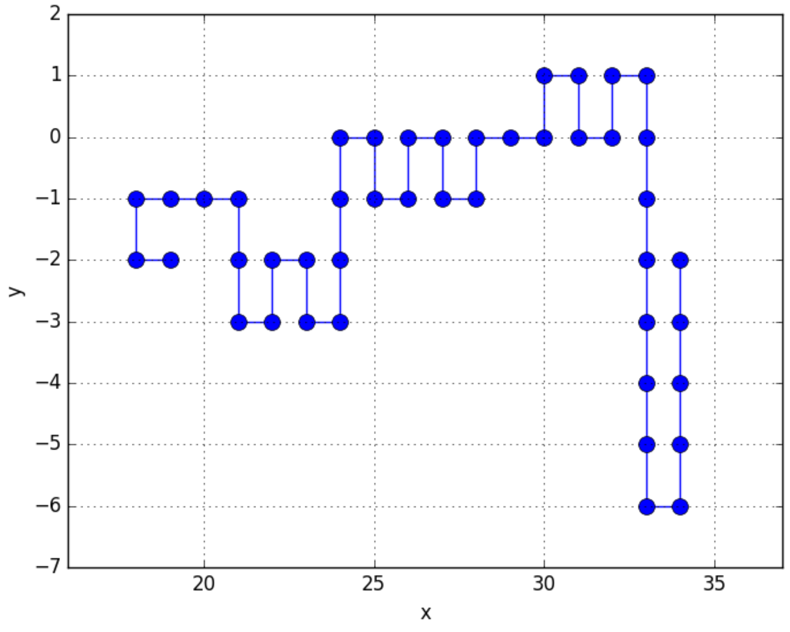

Figure 8: Amino acid chain shape at $t_7 = 10^7$

3 Analysis

As can be seen, the energy and the end-to-end distance decrease more or less exponentially as a function of time step, and the shape of the chain deviates more and more from its starting shape as time goes on. There are some random deviations from this trend that are caused by the random components underlying the system. The trend is simply a result of the rules regarding what would occur at each time step, as set forward at the outset: it is more likely than not that the movement of the amino acids results in a decrease in energy in the system, and the more convoluted the shape is, the less energy the system has, so it is to be expected that, starting from a perfectly straight shape, we end up with a chain that basically folds back upon itself. It is also the case that it becomes harder and harder for the energy of the system to keep decreasing as we reach lower and lower energies, simply because there are less moves available to the chain that are energetically advantageous. This simulation is a good demonstration that specific "environmental" impulses are able to shape the way a random system evolves over time, and that very small alterations in a system's structure are often able to slowly accumulate, in such a way that the end product is completely different than the starting product. Important developments in the sciences, like the theory of evolution by natural selection, for instance, rely on the truth of this notion. Figures 9-11 are meant to illustrate this concept. I ran the simulation again using a different seed for the RNG, and even though the final shape produced is different, the trends in the system's energy and end-to-end distance are the same as before. This results from the fact that the rules governing the system's evolution are insensitive to the specific shape of the chain; what matters in the end is whether the shape reduces the system's energy, and there are many different shapes that are able to do that.

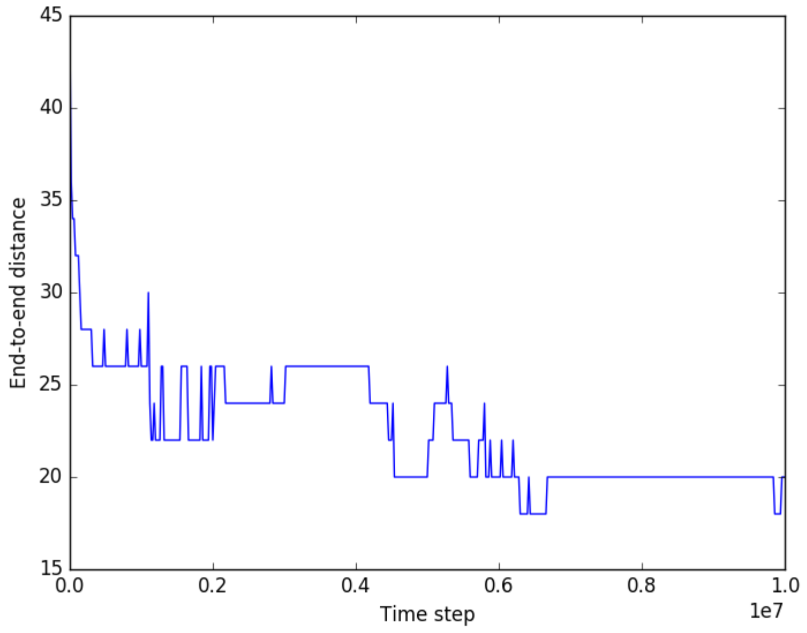

Figure 9: End-to-end distance vs time step for simulation #2

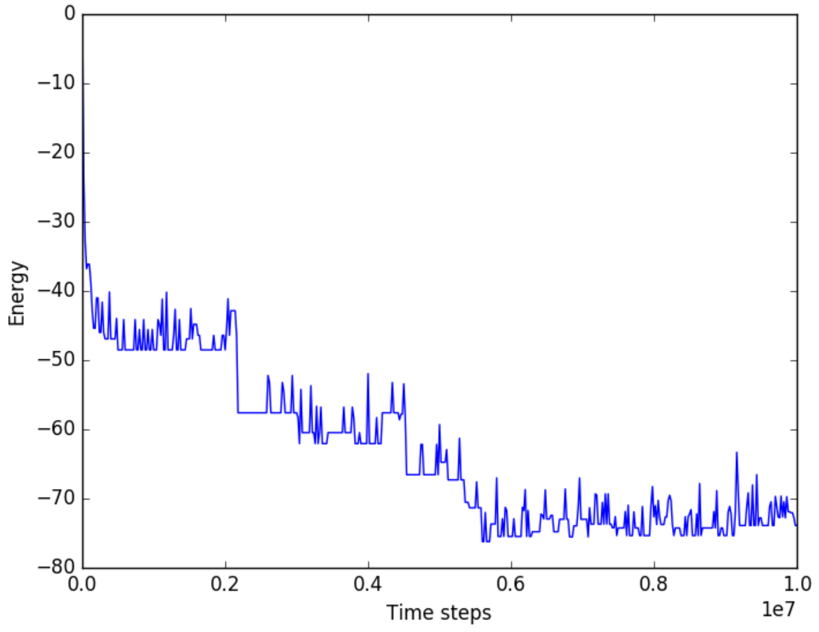

Figure 10: System energy vs time step for simulation #2

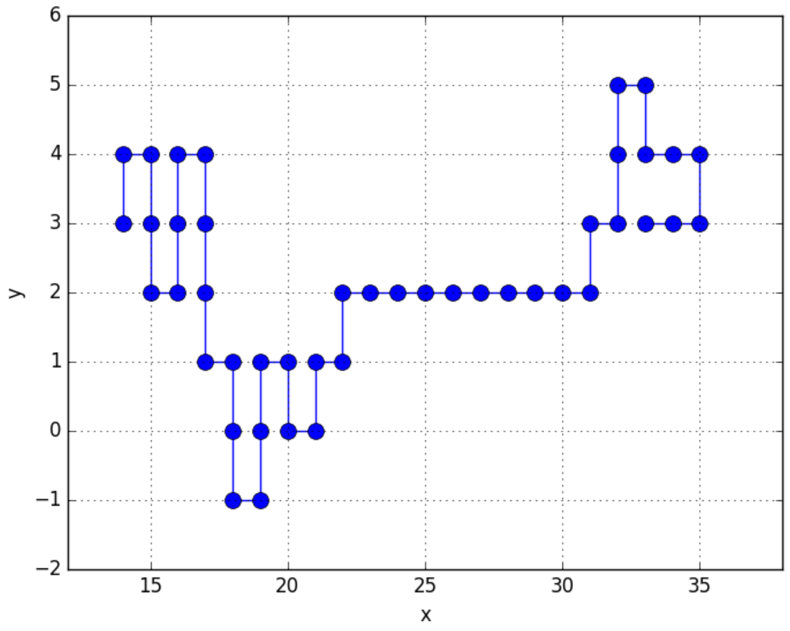

Figure 11: Amino acid chain shape at $t = 10^7$ for simulation #2

4 Source code

4.1 hw9a.cpp

```
/* AEP 4380 HW#9a
   Monte Carlo Calculations
   Run on core i7 with g++ 5.4.0 (Ubuntu)
   Gianfranco Grillo 11/20/2016
*/

#include <cstdlib>
#include <cmath>

#include <iostream>
#include <fstream>
#include <iomanip>
#include "arrayt.hpp"
#include "nr3.h"
#include "ran.h"

using namespace std;

int main() {
    int npts, npairs, nbins, n = 0, i;
    double rn, sqrsum;
    ofstream fp, sp;
    cout << "Enter␣number␣of␣points:␣";
    cin >> npts;
    cout << "Enter␣number␣of␣bins:␣";
    cin >> nbins;
    cout << "Enter␣number␣of␣pairs:␣";
    cin >> npairs;
    struct Ran myrand(17);
    arrayt<int> hist(nbins);
    arrayt<double> pairs(npairs,2), histranges(nbins+1);
    fp.open("histogram.dat");
    sp.open("pairs.dat");
    for (i = 0.0; i < nbins+1.0; i++) { histranges(i) = i; }
    for (i = 0; i < nbins; i++) { hist(i) = 0; }
    while (n < npts) {
        rn = myrand.doub();
        for (i = 0; i < histranges.n(); i++) {
            if (rn > (histranges(i)/nbins) && rn <= (histranges(i+1)/nbins)) {
                hist(i)++;
                break;
            }
        }
        n++;
    }
    sqrsum = 0.0;
    for (i = 0; i < npairs; i = i+2) {
        pairs(i, 0) = myrand.doub();
        pairs(i, 1) = myrand.doub();
        sp << pairs(i, 0) << setw(15) << pairs(i, 1) << endl;
    }
    for (i = 0; i < nbins; i++) {
        fp << hist(i) << endl;
    }
    fp.close();
    sp.close();
    return(EXIT_SUCCESS);
}
```

4.2 hw9b.cpp

```
/* AEP 4380 HW#9b
  Monte Carlo Calculations
  Run on core i7 with g++ 5.4.0 (Ubuntu)
  Gianfranco Grillo 11/20/2016
*/

#include <cstdlib>
#include <cmath>

#include <iostream>
#include <fstream>
#include <iomanip>
#include "arrayt.hpp"
#include "nr3.h"
#include "ran.h"

using namespace std;

int main() {
    int i, j, x, y, matsize = 20, naa = 45, N = 1e7, t = 0;
    struct Ran myrand(17);
    arrayt<double> enmat(matsize,matsize);
    arrayt<int> aalist(naa,3), aalistcopy(naa, 3), ranaa(naa), ranmov(4);
    double Emin = -7.0, Emax = -2.0, rn, a, b, e0, e1, R, p, deltae;
    double energy(arrayt<int>&, arrayt<double>&);
    double sep(int&, int&, int&, int&);
    void aamodifier(arrayt<int>&, int&, int, arrayt<int>&, arrayt<int>&);
    ofstream fp, sp;
    fp.open("data1.dat");
    sp.open("data2.dat");
    a = Emin/(Emax - Emin);
    b = Emax - Emin;
    for (i = 0; i < matsize; i++) for (j = 0; j < i+1; j++) { // initialize energy
        matrix
        rn = myrand.doub();
        enmat(i, j) = (rn + a)*b;  // need to adjust random number so it is in
            range Emin < rn < Emax
        if (i != j) { enmat(j, i) = enmat(i, j); }
    }
    for (i = 0; i < naa; i++) { // initialize amino acids horizontally
        aalist(i,0) = i; // initial x-coordinate
        aalist(i,1) = 0; // initial y-coordinate
        aalist(i,2) = (myrand.int64() % 20); // type of amino acid
    }
    for (i = 0; i < naa; i++) { ranaa(i) = i; }
    for (i = 0; i < 4; i++) { ranmov(i) = i; }
    e0 = energy(aalist, enmat);
    while (t < 1e7+1) {
        if (t % 20000 == 0) {
            fp << t << setw(15) << abs(aalist(0, 0) - aalist(naa-1, 0)) + abs(
                aalist(0, 1) - aalist(naa-1, 1)) << setw(15) << e0 << endl; // for
                some reason, trying to call sep() here results in gibberish
        }
        if (t == 1e4 || t == 1e5 || t == 1e6 || t == 1e7) {
            for (i = 0; i < naa; i++) {
                sp << t << setw(15) << aalist(i, 0) << setw(15) << aalist(i, 1) <<
                    endl;
            }
        }
        aalistcopy = aalist;
        aamodifier(aalist, naa, myrand.int64(), ranaa, ranmov);
        e1 = energy(aalist, enmat);
        deltae = e1 - e0;
        if (deltae <= 0) { e0 = e1; }
        else {
            p = exp(-deltae);
            R = myrand.doub();
            if (p <= R) { aalist = aalistcopy; }
            else { e0 = e1; }
        }
        t++;
    }
    return(EXIT_SUCCESS);
}

double energy(arrayt<int>& aalist, arrayt<double>& enmat) {
    int naa = aalist.n1(), i, j, x1, x2, y1, y2, type1, type2;
    double energy = 0.0;
    int sep(int&, int&, int&, int&);
    for (i = 0; i < naa; i++) for (j = 0; j < i; j++) {
        if (j != i+1 && j != i-1) { // no need to check covalent bonds
            x1 = aalist(i, 0);
            x2 = aalist(j, 0);
            y1 = aalist(i, 1);
            y2 = aalist(j, 1);
            if (sep(x1, x2, y1, y2) == 1) {
                type1 = aalist(i, 2);
                type2 = aalist(j, 2);
                energy += enmat(type1, type2);
            }
        }
    }
    return energy;
}

int sep(int& x1, int& x2, int& y1, int& y2) {
    int diff = (abs(x1 - x2) + abs(y1 - y2));
    return diff;
}

void aamodifier(arrayt<int>& aalist, int& naa, int rn2, arrayt<int>& ranaa, arrayt<
   int>& ranmov) {
    bool inarray(arrayt<int>&, int&, int&), allowed(int&, int&, int&, int&, int&,
       int&);
    void randomize(arrayt<int>&, int&);
    int p1x, p1y, p2x, p2y, newx, newy, oldx, oldy, i, j, aan, rn;
    randomize(ranaa, rn2);
    randomize(ranmov, rn2);
    for (i = 0; i < naa; i++) for (j = 0; j < 4; j++) {
        aan = ranaa(i);
        rn = ranmov(j);
        oldx = aalist(aan, 0);
        oldy = aalist(aan, 1);
        if (rn == 0) {
            newx = oldx - 1;
            newy = oldy + 1;
            if (aan == 0) { // special case 1
                p1x = aalist(aan + 1, 0);
                p1y = aalist(aan + 1, 1);
                if (inarray(aalist, newx, newy) == false && sep(newx, p1x, newy,
                   p1y) == 1) {
                    aalist(aan, 0) = newx;
                    aalist(aan, 1) = newy;
                    goto end_loop;
                }
            }
            if (aan == naa-1) { // special case 2
                p1x = aalist(aan - 1, 0);
                p1y = aalist(aan - 1, 1);
                if (inarray(aalist, newx, newy) == false && sep(newx, p1x, newy,
                   p1y) == 1) {
                    aalist(aan, 0) = newx;
                    aalist(aan, 1) = newy;
                    goto end_loop;
                }
            }
            else {
                p1x = aalist(aan + 1, 0);
                p1y = aalist(aan + 1, 1);
                p2x = aalist(aan - 1, 0);
                p2y = aalist(aan - 1, 1);
                if (inarray(aalist, newx, newy) == false && allowed(newx, newy, p1x
                   , p1y, p2x, p2y)) {
```

```
                aalist(aan, 0) = newx;
                aalist(aan, 1) = newy;
                goto end_loop;
            }
        }
    }
    if (rn == 1) {
        newx = oldx + 1;
        newy = oldy + 1;
        if (aan == 0) { // special case 1
            p1x = aalist(aan + 1, 0);
            p1y = aalist(aan + 1, 1);
            if (inarray(aalist, newx, newy) == false && sep(newx, p1x, newy,
                p1y) == 1) {
                aalist(aan, 0) = newx;
                aalist(aan, 1) = newy;
                goto end_loop;
            }
        }
        if (aan == naa-1) { // special case 2
            p1x = aalist(aan - 1, 0);
            p1y = aalist(aan - 1, 1);
            if (inarray(aalist, newx, newy) == false && sep(newx, p1x, newy,
                p1y) == 1) {
                aalist(aan, 0) = newx;
                aalist(aan, 1) = newy;
                goto end_loop;
            }
        }
        else {
            p1x = aalist(aan + 1, 0);
            p1y = aalist(aan + 1, 1);
            p2x = aalist(aan - 1, 0);
            p2y = aalist(aan - 1, 1);
            if (inarray(aalist, newx, newy) == false && allowed(newx, newy, p1x
                , p1y, p2x, p2y)) {
                aalist(aan, 0) = newx;
                aalist(aan, 1) = newy;
                goto end_loop;
            }
        }
    }
    if (rn == 2) {
        newx = oldx + 1;
        newy = oldy - 1;
        if (aan == 0) { // special case 1
            p1x = aalist(aan + 1, 0);
            p1y = aalist(aan + 1, 1);
            if (inarray(aalist, newx, newy) == false && sep(newx, p1x, newy,
                p1y) == 1) {
                aalist(aan, 0) = newx;
                aalist(aan, 1) = newy;
                goto end_loop;
            }
        }
        if (aan == naa-1) { // special case 2
            p1x = aalist(aan - 1, 0);
            p1y = aalist(aan - 1, 1);
            if (inarray(aalist, newx, newy) == false && sep(newx, p1x, newy,
                p1y) == 1) {
                 aalist(aan, 0) = newx;
                 aalist(aan, 1) = newy;
                 goto end_loop;
            }
        }
        else {
            p1x = aalist(aan + 1, 0);
            p1y = aalist(aan + 1, 1);
            p2x = aalist(aan - 1, 0);
            p2y = aalist(aan - 1, 1);
            if (inarray(aalist, newx, newy) == false && allowed(newx, newy, p1x
                , p1y, p2x, p2y)) {
                 aalist(aan, 0) = newx;
                 aalist(aan, 1) = newy;
                 goto end_loop;
            }
        }
    }
    if (rn == 3) {
        newx = oldx - 1;
        newy = oldy - 1;
        if (aan == 0) { // special case 1
            p1x = aalist(aan + 1, 0);
            p1y = aalist(aan + 1, 1);
            if (inarray(aalist, newx, newy) == false && sep(newx, p1x, newy,
                p1y) == 1) {
                 aalist(aan, 0) = newx;
                 aalist(aan, 1) = newy;
                 goto end_loop;
            }
        }
        if (aan == naa-1) { // special case 2
            p1x = aalist(aan - 1, 0);
            p1y = aalist(aan - 1, 1);
            if (inarray(aalist, newx, newy) == false && sep(newx, p1x, newy,
                p1y) == 1) {
                 aalist(aan, 0) = newx;
                 aalist(aan, 1) = newy;
                 goto end_loop;
            }
        }
        else {
            p1x = aalist(aan + 1, 0);
            p1y = aalist(aan + 1, 1);
```

```
                p2x = aalist(aan - 1, 0);
                p2y = aalist(aan - 1, 1);
                if (inarray(aalist, newx, newy) == false && allowed(newx, newy, p1x
                    , p1y, p2x, p2y)) {
                    aalist(aan, 0) = newx;
                    aalist(aan, 1) = newy;
                    goto end_loop;
                }
            }
        }
    }
    end_loop:
    return;
}

bool inarray(arrayt<int>& aalist, int& newx, int& newy) {
    int size = aalist.n1(), i, x, y;
    bool r = false;
    for (i = 0; i < size; i++) {
        x = aalist(i, 0);
        y = aalist(i, 1);
        if (newx == x && newy == y) { r = true; }
    }
    return r;
}

bool allowed(int& newx, int& newy, int& p1x, int& p1y, int& p2x, int& p2y) {
    int sep(int&, int&, int&, int&);
    int sep1, sep2;
    sep1 = sep(newx, p1x, newy, p1y);
    sep2 = sep(newx, p2x, newy, p2y);
    if (sep1 > 1 || sep2 > 1) { return false; }
    else { return true; }
}

void randomize(arrayt<int>& list, int& rn) {
    int size, i, rn1, rn2;
    struct Ran myrand2(rn);
    size = list.n();
    for (i = 0; i < size; i++) {
        rn1 = myrand2.int64() % size;
        rn2 = myrand2.int64() % size;
        SWAP(list(rn1), list(rn2));
    }
    return;
}
```

References

[1] W. H. Press, S. A. Teukolsky, W. T. Vetterling and B. P. Flannery *Numerical Recipes, The Art of Scientific Computing, 3rd edit.*, Camb. Univ. Press 2007.