AEP4380 HW9
Monte Carlo Calculations

Dan Girshovich

April 18, 2013

1 Overview

This document details the results of the included C++ program, which uses a simplified model of a 2D protein structure to find low energy configurations via a Monte Carlo method. This acts as a demonstration use case for pseudorandom number generators in numerical computations.

2 Pseudorandom Number Generation

2.1 Overview

Monte Carlo methods require random sampling, which is most commonly provided from pseudorandom number generator. Here, the `doub` function of the `Ran` generator on page 342 of Numerical Recipes [1] is used as the generator and it is seeded with the system time. This function should (seemingly at random) return a double between 0 and 1 uniformly.

2.2 Test Runs

As a quick test of the randomness of the generator, a normalized histogram of the first $100,000$ values returned is shown in Figure 1. Also, Figure 2 shows the result of generating $10,000$ pairs of values. The flatness of the histogram and the evenness of the pair points suggests that the generator reliably produces uniformly distributed values.

3 Protein Folding Problem

3.1 Overview

A simplified model of protein folding is discussed section 11.1 of Giordano [2]. The idea is to model all bonds as having equal lengths and right-angle bends. This allows each amino acid to be represented as a point on a cartesian grid and each bond as a vertical or horizontal edge between the points. To simplify further, the forces between non-bonded acids

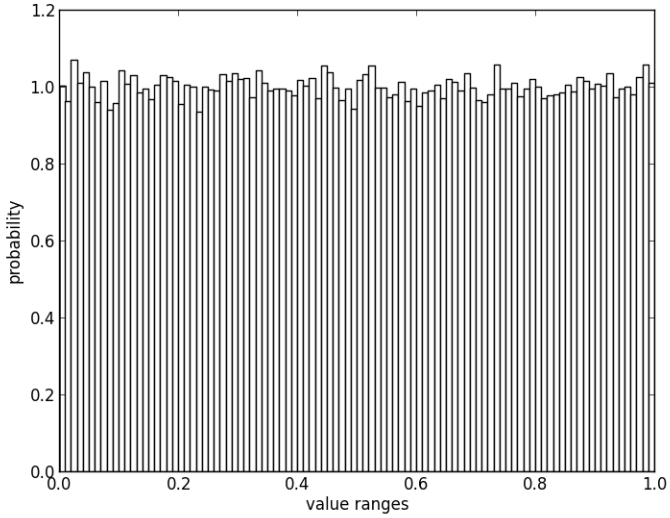


Figure 1: Normalized histogram of $100,000$ pseudorandom values with 100 bins.

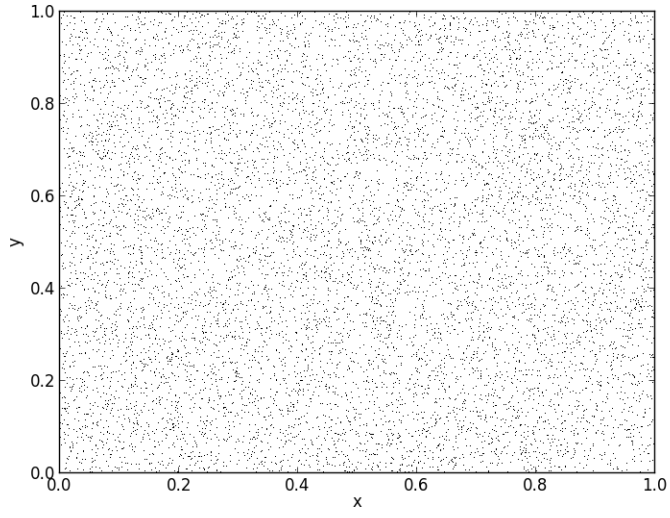


Figure 2: Point plot of $10,000$ pseudorandom value pairs.

are approximated to be only between nearest neighbors (vertically or horizontally adjacent acids). In this case, the energy of the protein is given by:

$$E_{tot} = \sum_{\text{all pairs}} E_{t(i),t(j)}\delta_{ij}.$$

Here, $E_{t(i),t(j)}$ is the interaction energy between amino acids of type $t(i)$ and $t(j)$ and $\delta_{ij} = 0$ unless the acids i and j are nearest neighbors, in which case $\delta_{ij} = 1$. The final simplification is that all bond energies are assumed to be constant. So, as the equation states, the only contribution to the energy of a configuration is that from the interaction between nearest neighbors.

3.2 Monte Carlo Algorithm

A low energy configuration is found using this simple model and a Monte Carlo method. This method involves performing exploratory random folds on a (initially straight) chain of acids. If a fold reduces the total energy, it is accepted (its effects are kept). Otherwise, it is only accepted with a probability that decreases exponentially with the size of the proposed energy increase:

$$P = e^{\frac{-\Delta E}{k_B T}}$$

Here, ΔE is the energy increase that performing the fold would cause, k_B is Boltzmann's constant, and T is temperature. This algorithm has the net effect of driving the chain towards low total energy configurations. The fact that energy can sometimes increase models the thermal fluctuations in the system. Importantly, this helps prevent the algorithm from getting stuck in energy states that are only locally minimal (since multiple folds may be needed to reach a less energetic state from them).

4 Monte Carlo Results

4.1 Overview

The following results were obtained from running the above algorithm 10^7 times at a temperature that produces $k_B T = 1$ for simplicity. The chain was initialized as a horizontal line of 40 acids and the types of acids were chosen at random from a total of 20. The energies corresponding to interactions between types were initialized to be random values between $E_{min} = -5$ and $E_{max} = -2$. All energies have units of bonding energy.

4.2 Energy

The total energy of the configuration during the simulation is plotted in Figure 3. As expected, the net effect of the algorithm is to reduce the total energy, but not strictly. The rate at which the energy levels off to the minimum values of ≈ -80 is roughly exponential.

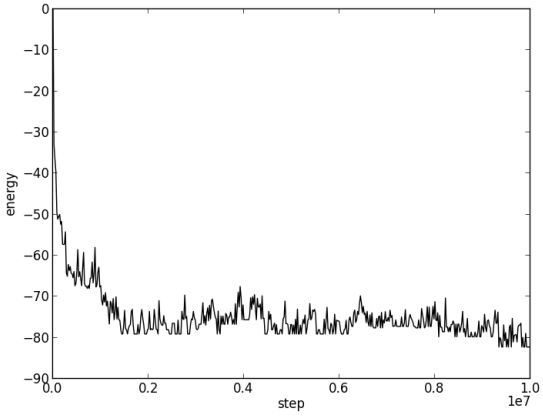


Figure 3: Total energy of the chain as the algorithm progresses. The energies are in units of bond energy.

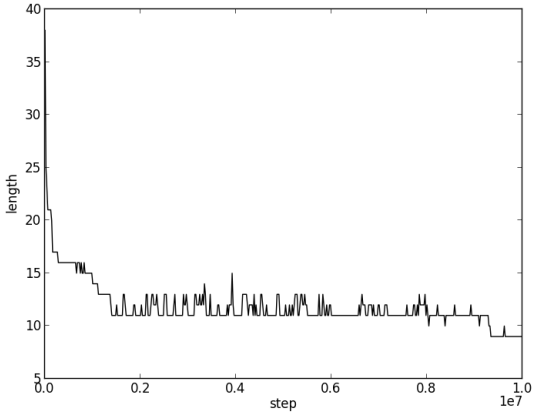


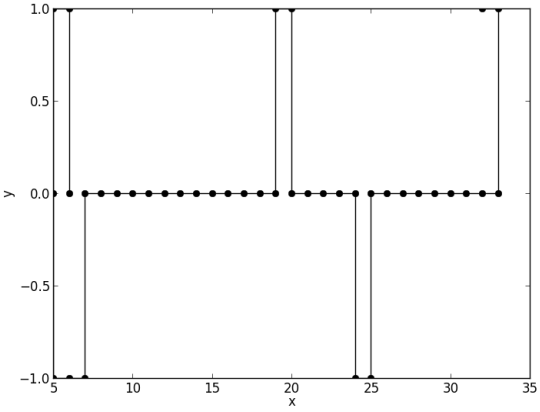
Figure 4: Horizontal width of the chain as the algorithm progresses. The lengths are in units of bond length.

4.3 Length

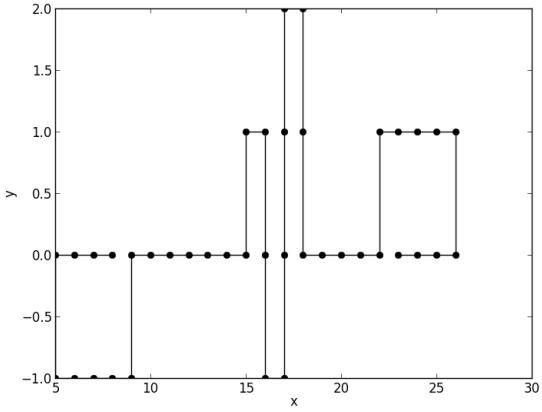
Figure 4 shows the total horizontal length of the chain as the algorithm progresses. This gives a good picture of when the significant folds were done, and nicely matches the energy plot. It is expected that energy would decrease as width decreases, since interaction energies are negative, and there tends to be more interaction in tighter packed chains.

4.4 Chain Structure

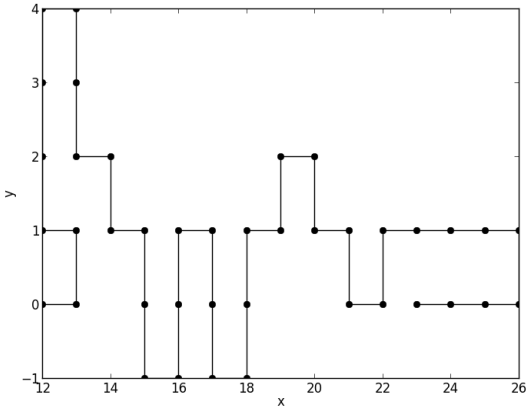
The chain structures at various time steps are shown in Figure 5. The final configuration is specific to the random types and energies chosen for that run. Still, the trend of moving towards more highly packed configurations is evident.



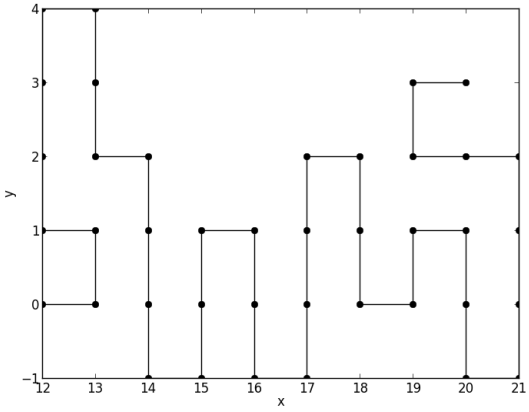
(a) 10^4



(b) 10^5



(c) 10^6



(d) 10^7

Figure 5: Protein structure at 10^4, 10^5, 10^6, and 10^7 time steps.

5 Source Code

5.1 Overview

The code is divided into three parts: the main drivers, a chain class, and a acid class. The classes help keep the more complex parts of the code (like the folding or main Monte Carlo algorithm) short and straightforward by hiding many details.

5.2 Implementation

5.2.1 Main

```
/*
AEP 4380 HW #9
Dan Girshovich
4/18/13
Monte Carlo Calculations
Compile with: g++ -std=c++0x -O2 -o gen_data hw9.cpp
Tested on Mac OSX 10.8.2 with a Intel Core 2 Duo
*/

#include "chain.cpp"

// performs num_steps + 1 monte carlo steps
void monte_carlo(int num_steps) {
    Chain chain;
    ofstream of1, of2;
    of1.open("energy.dat");
    of2.open("length.dat");
    for (int i = 0; i <= num_steps; i++) {
        Chain new_chain(chain);
        new_chain.fold();
        double dE = new_chain.energy - chain.energy;
        if (dE <= 0 || rnd.doub() < exp(-1 * dE)) {
            chain = new_chain;
        }
        if ((i % ((num_steps / 500) + 1)) == 0) {
            of1 << i << " " << chain.energy << std::endl;
            of2 << i << " " << chain.get_length() << std::endl;
        }
        if (i == 1e4 || i == 1e5 || i == 1e6 || i == 1e7) {
            chain.write((int) log10(i));
        }
    }
    of1.close();
    of2.close();
}

void gen_hist_data() {
    ofstream of;
    of.open("hist.dat");
    for (int i = 0; i < 100000; i++) {
```

```
        of << rnd.doub() << endl;
    }
    of.close();
}

void gen_points_data() {
    ofstream of;
    of.open("points.dat");
    for (int i = 0; i < 10000; i++) {
        of << rnd.doub() << " " << rnd.doub() << endl;
    }
    of.close();
}

int main() {
    gen_hist_data();
    gen_points_data();
    monte_carlo(1e7);
}
```

5.2.2 Chain Class

```
#include "nr3.h"
#include "ran.h"
#define ARRAYT_BOUNDS_CHECK
#include "array.hpp"
#include "acid.cpp"

const int NUM_ACIDS = 40, NUM_TYPES = 20;
const double E_min = -5, E_max = -2;

class Chain {
public:
    double energy;
    array<Acid> arr;
    array<double> E;
    Chain();
    Chain(Chain &c);
    void fold();
    int get_length();
    void write(int i);
private:
    void init_acids();
    void init_energies();
    double calc_energy();
    bool contains(int x, int y);
    bool try_fold();
};

Chain::Chain() : E(NUM_TYPES, NUM_TYPES), arr(NUM_ACIDS) {
    init_acids();
    init_energies();
    energy = calc_energy();
}
```

```
Chain::Chain(Chain &c): E(c.E), arr(c.arr), energy(c.energy) {}

// initializes the chain in a line with random types
void Chain::init_acids() {
    for (int i = 0; i < NUM_ACIDS; i++) {
        int t = (int) (rnd.doub() * NUM_TYPES);
        Acid a(i, 0, t); // horizontal line
        arr(i) = a;
    }
}

// randomly sets the symmetric energy matrix
void Chain::init_energies() {
    for (int i = 0; i < NUM_TYPES; i++) {
        for (int j = i; j < NUM_TYPES; j++) {
            E(i, j) = E_min + (E_max - E_min) * rnd.doub();
            E(j, i) = E(i, j);
        }
    }
}

// returns the enery of the chain using the E matrix
double Chain::calc_energy() {
    double sum = 0.0;
    for (int i = 0; i < NUM_ACIDS - 1; i++) {
        Acid a = arr(i);
        for (int j = i + 1; j < NUM_ACIDS; j++) {
            Acid b = arr(j);
            bool bonded = abs(i - j) == 1;
            if (!bonded) {
                if (a.is_neighbor(b)) {
                    sum += E(a.t, b.t);
                }
            }
        }
    }
    return sum;
}

// returns true if the chain contains an acid at point (x, y)
bool Chain::contains(int x, int y) {
    for (int i = 0; i < NUM_ACIDS; i++) {
        Acid a = arr(i);
        if (a.x == x && a.y == y) return true;
    }
    return false;
}

// tries to make a random fold and updates the chain and energy if successful
// returns true if successful, false otherwise
bool Chain::try_fold() {
    // randomly pick an acid
    int i = (int) (NUM_ACIDS * rnd.doub());
```

```
    Acid new_a(arr(i));
    // randomly choose a perturbation to try
    new_a.perturb();
    // check availability
    if (contains(new_a.x, new_a.y)) return false;
    // check left neighbor correctly bonded
    if (i > 0) {
        Acid left_n = arr(i - 1);
        if (!new_a.is_neighbor(left_n)) return false;
    }
    // check right neighbor correctly bonded
    if (i < NUM_ACIDS - 1) {
        Acid right_n = arr(i + 1);
        if (!new_a.is_neighbor(right_n)) return false;
    }
    // all checks passed
    arr(i) = new_a;
    energy = calc_energy();
    return true;
}

void Chain::fold() {
    while (!try_fold());
}

int Chain::get_length() {
    int min = 1000, max = -1000;
    for (int i = 0; i < NUM_ACIDS; i++) {
        int x = arr(i).x;
        if (x < min) min = x;
        if (x > max) max = x;
    }
    return max - min;
}

void Chain::write(int i) {
    stringstream fname;
    fname << "chain" << i << ".dat";
    ofstream of;
    of.open(fname.str().c_str());
    for (int i = 0; i < NUM_ACIDS; i++) {
        of << arr(i).x << " " << arr(i).y << std::endl;
    }
    of.close();
}
```

5.2.3 Acid Class

```
#include <cstdio>
#include <cstdlib>
#include <cmath>
#include <ctime>

Ran rnd(time(NULL));
```

```
class Acid {
public:
    int x, y, t;
    Acid(int x, int y, int t): x(x), y(y), t(t) {}
    Acid(): x(-1), y(-1), t(-1) {} // for use as array element
    Acid(Acid &a): x(a.x), y(a.y), t(a.t) {}
    bool is_neighbor(Acid a);
    void perturb();
};

// true if the acid and acid a are vertically or horizontally adjacent
bool Acid::is_neighbor(Acid a) {
    return (abs(a.x - x) + abs(a.y - y)) == 1;
}

// randomly returns positive or negative one
int get_offset() {
    if (rnd.doub() < 0.5) return -1;
    return 1;
}

// moves the acid to a positional diagonally adjacent to it
void Acid::perturb() {
    x += get_offset();
    y += get_offset();
}
```

References

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

[2] N. J. Giordano, and H. Nakanishi, *Computational Physics*. Prentice-Hall, 2nd Edition, 2006.