

Self-Avoiding Walks and Polymers

Polymers in Solvent

A polymer is a chain molecule with a large number $n + 1$ of monomers joined by n strong covalent bonds. Protein molecules [1] are polymer chains made from 20 types of amino acid.

In a solvent, the polymer chain can fold into complex 3-dimensional shapes. These configurations are determined by weak Van der Waals forces and hydrogen bonds between monomers, and also by interactions with solvent molecules. The folded shape of protein molecules determines their functional capabilities.

Polymer configuration and random walks

Can we model polymer chain configurations by random walks in 3-D? Start with a single monomer and add successive monomers with bonds oriented in random directions.

This type of random walk is not the same as the simple random walk studied in the previous lecture. It cannot be a Markov chain because monomers cannot get closer than their molecular diameter – *excluded volume effect* – the probability of the next step depends on the locations of all prior monomers, i.e., on the whole history of the walk!

This type of random walk is called a self-avoiding walk.

Self-avoiding walks in one dimension are trivial. There are only two different self-avoiding walks with fixed step size in 1-D. If the first move is to the left, every subsequent step is to the left. If the first move is to the right, the walk continues to the right. The displacement of the walker is motion with constant speed

$$|x| = n \propto t, \quad (1)$$

which is *ballistic motion*, compared with diffusive motion for the simple walk

$$\sqrt{\langle x_n^2 \rangle} = \sqrt{n} \propto \sqrt{t}. \quad (2)$$

Non-trivial SAW behavior occurs in 2 dimensions or greater

$$\sqrt{\langle r^2 \rangle} \sim At^\nu, \quad (3)$$

where the Flory exponent ν is intermediate between the diffusive and ballistic values

$$\frac{1}{2} < \nu < 1. \quad (4)$$

The Flory exponent determines the mean end-to-end size of polymer molecules.

Self-Avoiding Walk in Two Dimensions

Generating self-avoiding walks is tricky! Consider a self-avoiding walk on a 2-D square lattice. The walker must step North, South, East or West with equal probability. At the same time the walker must avoid previously visited locations. Unfortunately, these two requirements are incompatible! A walker who steps with equal probability NSEW cannot avoid previously visited locations. The first step has 4 allowed directions, NSEW. Every subsequent step has 3, 2, 1, or 0 allowed directions.

The fundamental problem is to ensure that every SAW chain in an equilibrium ensemble occurs with the same probability.

Simplistic Self-Avoiding Walk Program

The following program has the walker step NSEW with equal probabilities, and simply discards failed walks.

_____ Program 1: <http://www.physics.buffalo.edu/phy410-505/topic2/sawalk.cpp> _____

```
#include <cmath>
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <vector>
using namespace std;

#include "random.hpp"
using namespace cpl;

struct Site {          // object to represent lattice site
    int x;             // x coordinate
    int y;             // y coordinate
};

int main() {

    cout << " Self-Avoiding Walks on a Square Lattice\n"
         << " -----\n"
         << " Enter number of steps in walk: ";
    int nSteps, nWalks;
    cin >> nSteps;
    cout << " Enter number of walks to generate: ";
    cin >> nWalks;

    int walks = 0;
    int failedWalks = 0;
    double r2av = 0;
    double r4av = 0;

    // create a random number generator object
    Random r;
    r.set_seed_time();
    cout << " Using " << r.get_algorithm()
         << " and seed " << r.get_seed() << endl;

    // generate walks
    while (walks < nWalks) {
        vector<Site> sites;          // set of occupied lattice sites
        Site s;
        s.x = 0;
        s.y = 0;
        sites.push_back(s);
        bool walkFailed = false;
```

```

// loop over desired number of steps
for (int step = 0; step < nSteps; step++) {

    // take a random step
    double d = r.rand();
    if      (d < 0.25) ++s.x;  // step East
    else if (d < 0.50) ++s.y;  // step North
    else if (d < 0.75) --s.x;  // step West
    else          --s.y;  // step South

    // check whether the site is occupied
    bool occupied = false;
    for (int i = 0; i < sites.size(); i++) {
        if (s.x == sites[i].x && s.y == sites[i].y) {
            occupied = true;
            break;
        }
    }
    if (occupied) {
        walkFailed = true;
        break;
    }

    sites.push_back(s);
}

if (walkFailed) {
    ++failedWalks;
    continue;
}

double r2 = s.x * s.x + s.y * s.y;
r2av += r2;
r4av += r2 * r2;
++walks;
}

r2av /= nWalks;
r4av /= nWalks;
double stdDev = sqrt(r4av - r2av * r2av);
double totalWalks = nWalks + failedWalks;
double failedPercent = failedWalks / totalWalks * 100.0;
cout << " Mean square distance <r^2> = " << r2av << "\n"
     << " Standard deviation          = " << stdDev << "\n"
     << " Percentage failed walks      = " << failedPercent
     << endl;
}

```

Running the program shows that it is extremely inefficient. The fraction of discarded walks increases

exponentially with length n

$$\frac{\text{Walks Generated}}{\text{Total Number of Attempts}} \sim e^{-\lambda n}, \quad (5)$$

where λ is a positive *attrition constant*.

The Reptation Method

The reptation model of polymer diffusion was developed by Pierre de Gennes who won the Nobel Prize in 1991 for his work on liquid crystals and polymers.

Polymers form an entangled mass of chains. De-Gennes assumed that any polymer is confined to a tube, which snakes through the tangle. Thermal fluctuations cause the polymer chain to *reptate* (creep) through the tube in the manner of a reptile.

Wall and Mandel [2] introduced a “slithering snake” model to generate equilibrium polymer ensembles. Their model avoids the attrition problem.

Reptation Algorithm

The reptation algorithm implements the slithering snake model.

Algorithm 1Reptation

- Start with any chain with a given number n of links.
 - Repeat
 - Choose one end of the chain at random
 - Choose at random one of the 3 possible reptation directions, (one forward, and two sideways)
 - If the site in the randomly chosen direction is not one of the interior sites in the chain (it can be the other end site)
 - remove the site at the other end of the chain
 - add this site to the chain
 - count the new configuration as the next in the ensemble
 - Otherwise
 - retain the current configuration as the next in the ensemble
-

Note that NO configuration is discarded in this algorithm. It can be shown that the algorithm correctly generates an equilibrium ensemble EXCEPT for double cul-de-sac configurations, which cannot be generated by reptation. A chain end is in a cul-de-sac (dead end) if all its neighbors are occupied. A *double* cul-de-sac configuration is one in which *both* ends are in cul-de-sacs and are not adjacent, as shown Fig. 1. If the two ends are not adjacent, the reptile can slither out of the configuration. Double cul-de-sacs are rare

C++ Reptation Program

The following program implements the reptation algorithm on a two-dimensional square lattice.


```

// strict weak-ordering comparison operator required by set
bool operator< (const Site& s) const {
    const int maxDiff = 9999; // must be larger than x or y difference
    return (x - s.x) + maxDiff * (y - s.y) < 0;
}
};

deque<Site> snake; // double-headed reptile
set<Site> occupiedSites; // set of occupied sites

bool occupied(Site s) { // return true if s is occupied
    return occupiedSites.find(s) != occupiedSites.end();
}

void clear() { // remove all sites
    snake.clear();
    occupiedSites.clear();
}

void addBack(Site s) { // add s to back of reptile
    snake.push_back(s);
    occupiedSites.insert(s);
}

void addFront(Site s) { // add s to back of reptile
    snake.push_front(s);
    occupiedSites.insert(s);
}

void removeBack() { // remove back end of reptile
    occupiedSites.erase(snake.back());
    snake.pop_back();
}

void removeFront() { // remove front end of reptile
    occupiedSites.erase(snake.front());
    snake.pop_front();
}

```

Initializing a walk

Next, constants and a function to initialize a random walk with a given number of steps are defined.

_____ Program 2: <http://www.physics.buffalo.edu/phy410-505/topic2/reptation.cpp> _____

```

const int EAST = 0, NORTH = 1, WEST = 2, SOUTH = 3, DIRECTIONS = 4;

const int // initial configuration choices

```

```

STAIR = 1,          // East-North random staircase
COIL  = 2,          // tight coil East, North, West, West, South, ...
LINE  = 3;          // straight line East

void createSnake(   // make a snake with
  int steps,        // this number of segments
  int config = LINE) // and this initial configuration
{
  clear();          // remove all sites

  Site s;
  s.x = s.y = 0;
  addFront(s);

  for (int step = 1; step <= steps; step++) {

    int stp = 0, dir = EAST; // initialize variables to construct coil

    switch (config) {
    case STAIR:        // add randomly East or North
      r.rand() < 0.5 ? ++s.x : ++s.y;
      break;
    case COIL:         // add in sequence E,N,W,W,S,S,E,E,E,N,N,N,...
      while (stp < step)
        stp += ++dir / 2;
      switch ((dir + 2) % DIRECTIONS) {
        case EAST : ++s.x; break;
        case NORTH : ++s.y; break;
        case WEST  : --s.x; break;
        case SOUTH : --s.y; break;
      }
      break;
    case LINE:         // add East
    default:           // also the default
      ++s.x;
    }

    addFront(s);
  }
}

```

Choosing the next site

The following function chooses a random site for the next attempted reptation step

_____ Program 2: <http://www.physics.buffalo.edu/phy410-505/topic2/reptation.cpp> _____

```

Site randomAllowed( // return a random allowed site

```

```

Site head,           // adjacent to this head site
Site neck)          // excluding this neck site
{
    deque<Site> allowed;

    // find the 3 allowed directions and add site to allowed deque

    for (int direction = EAST; direction < DIRECTIONS; direction++) {

        Site s;
        s.x = head.x;
        s.y = head.y;

        switch (direction) {
        case EAST:
            if ( !(head.x == neck.x - 1 && head.y == neck.y) ) {
                ++s.x;
                allowed.push_back(s);
            }
            break;
        case NORTH:
            if ( !(head.x == neck.x && head.y == neck.y - 1) ) {
                ++s.y;
                allowed.push_back(s);
            }
            break;
        case WEST:
            if ( !(head.x == neck.x + 1 && head.y == neck.y) ) {
                --s.x;
                allowed.push_back(s);
            }
            break;
        case SOUTH:
            if ( !(head.x == neck.x && head.y == neck.y + 1) ) {
                --s.y;
                allowed.push_back(s);
            }
            break;
        }
    }

    // choose and return a random allowed site
    return allowed[int(2.999999 * r.rand())];
}

```

The reptation algorithm

The following function implements the reptation algorithm using the functions defined above

```
bool reptate() {          // attempt random move and return true if succeeded

    if (snake.size() < 2)          // cannot reptate
        return false;

    Site head, neck, sNext;

    if (r.rand() < 0.5) {          // choose front end of snake
        head = snake[0];
        neck = snake[1];
        sNext = randomAllowed(head, neck);
        if (occupied(sNext))
            return false;
        removeBack();
        addFront(sNext);
    } else {                        // choose back end of snake
        int n = snake.size();
        head = snake[n - 1];
        neck = snake[n - 2];
        sNext = randomAllowed(head, neck);
        if (occupied(sNext))
            return false;
        removeFront();
        addBack(sNext);
    }

    return true;
}

double rSquared() {              // end-to-end size squared
    if (snake.size() < 2)
        return 0.0;
    double dx = snake.front().x - snake.back().x;
    double dy = snake.front().y - snake.back().y;
    return dx * dx + dy * dy;
}

int main() {

    cout << " Reptation Method for Self-Avoiding Walks on a Square Lattice\n"
         << " -----\n"
         << " Enter maximum number of steps in walk: ";
    int nSteps, nWalks;
    cin >> nSteps;
    cout << " Enter number of walks to generate: ";
    cin >> nWalks;
    cout << " Enter initial configuration 1 = stair, 2 = coil, 3 = line: ";
    int config;
    cin >> config;
```

```

// set random number generator seed
Random r;
r.set_seed_time();
cout << " Using " << r.get_algorithm()
    << " and seed " << r.get_seed() << endl;

ofstream file("reptation.data");
cout << " Steps   <r^2>          Std. Dev.   Success%   CPU secs" << endl;

for (int steps = 1; steps <= nSteps; steps++) {

    double r2sum = 0;
    double r4sum = 0;
    int success = 0;
    clock_t startTime = clock();

    createSnake(steps, config);

    for (int i = 0; i < nWalks; i++) {
        if (reptate())
            ++success;
        double r2 = rSquared();
        r2sum += r2;
        r4sum += r2 * r2;
    }

    clock_t endTime = clock();

    double r2av = r2sum / nWalks;
    double stdDev = sqrt(r4sum / nWalks - r2av * r2av);
    double successPercent = success / double(nWalks) * 100.0;
    double cpu = (endTime - startTime) / double(CLOCKS_PER_SEC);

    cout << right << setw(4) << steps           << "      "
        << left  << setw(10) << r2av             << "    "
        << left  << setw(10) << stdDev           << "    "
        << left  << setw(8)  << successPercent  << "    "
        << left  << setw(8)  << cpu             << '\n';
    file << steps << '\t' << r2av << '\t' << stdDev
        << '\t' << successPercent << '\t' << cpu << '\n';
}

file.close();
cout << " Data in file reptation.data" << endl;
}

```

Program output

Fig. 2 shows typical output from running the program.

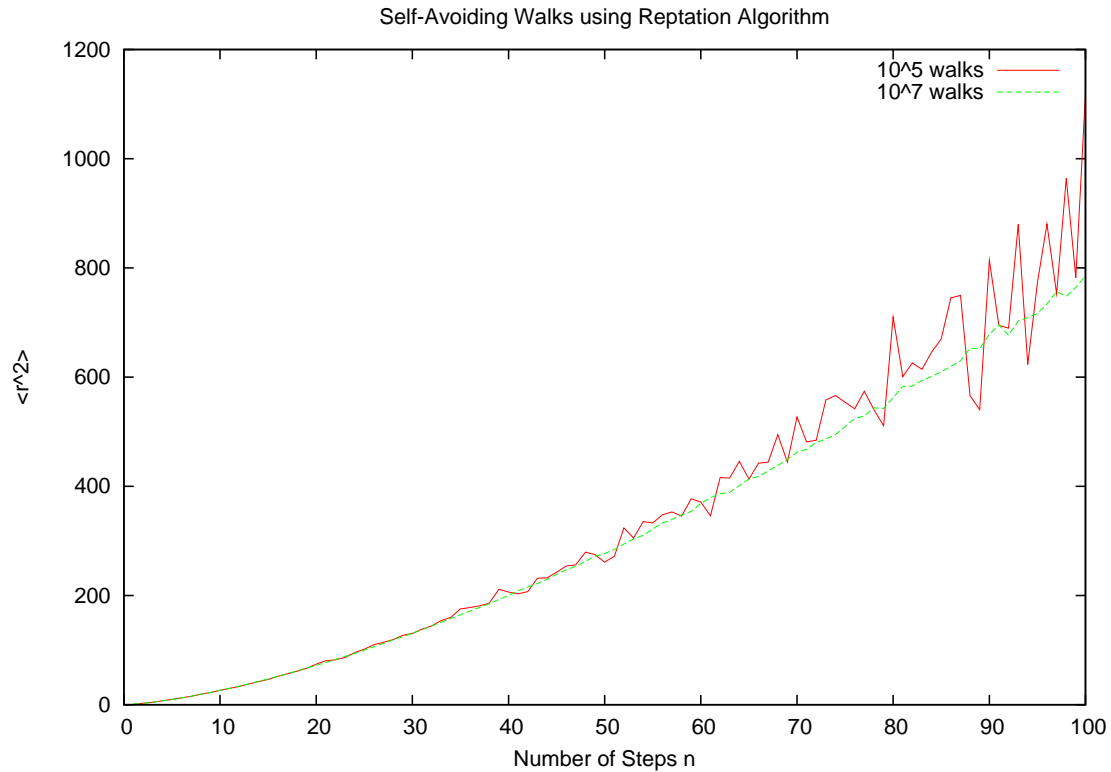


Figure 2: Note the dramatic growth of fluctuations with the length of the walk.

Homework Problem

Compare the CPU times required to generate samples of SAWs of various lengths using the simplistic and reptation algorithms. Use the reptation algorithm to measure the Flory exponent for SAWs on a 2-D lattice. Compare your result with the non-SAW random walk of the previous lecture and explain the difference.

References

- [1] RCSB Protein Data Bank Website: <http://www.rcsb.org/pdb/home/home.do>
- [2] F.T. Wall and F. Mandel, "Macromolecular dimensions obtained by an efficient Monte Carlo method without sample attrition", *J. Chem. Phys.* **63**, 4592 (1975), <http://link.aip.org/link/?JCPA6/63/4592/1>.