Statistics of Polymer Configuration

A polymer is a chain molecule with a large number \( n + 1 \) of monomers joined by \( n \) strong covalent bonds. Protein molecules 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, which is a Markov chain in which the next step is independent of the history of the walk. 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

\[ |x| = n \propto t , \]
which is \textit{ballistic motion}, compared with diffusive motion for the simple walk

\[ \sqrt{\langle x_n^2 \rangle} = \sqrt{n} \propto \sqrt{t}. \]

Non-trivial SAW behavior occurs in 2 dimensions or greater

\[ \sqrt{\langle r^2 \rangle} \sim At^\nu, \]

where the Flory exponent \( \nu \) is intermediate between the diffusive and ballistic values

\[ \frac{1}{2} < \nu < 1. \]

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

\textbf{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.

\textbf{Code for a Simplistic Self-Avoiding Walk Algorithm}
The following code models a self-avoiding random walk in two dimensions by having the walker step NSEW with equal probabilities, and simply discarding failed walks.

Running the code shows that the algorithm is extremely inefficient. The fraction of discarded walks increases exponentially with length \( n \)

\[
\frac{\text{Walks Generated}}{\text{Total Number of Attempts}} \sim e^{-\lambda n},
\]

where \( \lambda \) is a positive attrition constant.

This Self-avoiding Walk Java Applet demonstrates the problem.

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

inline double std_rand()
{
    return rand() / (RAND_MAX + 1.0);
}
```
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 n_steps, n_walks;
    cin >> n_steps;
    cout << " Enter number of walks to generate: ";
    cin >> n_walks;

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

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

// loop over desired number of steps
for (int step = 0; step < n_steps; step++) {
    // take a random step
    double d = std_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) {


```c++
walk_failed = true;
break;
}

sites.push_back(s);
}

if (walk_failed) {
    ++failed_walks;
    continue;
}

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

r2av /= n_walks;
r4av /= n_walks;
double stdDev = sqrt(r4av - r2av * r2av);
double totalWalks = n_walks + failed_walks;
double failedPercent = failed_walks / totalWalks * 100.0;
cout << " Mean square distance \langle r^2 \rangle = " << r2av << "\n"


```
<< " Standard deviation = " << stdDev << "\n"
<< " Percentage failed walks = " << failedPercent
<< endl;
```
Exact Enumeration of Self-avoiding Walks

Finding the exact number of self-avoiding walks is a challenging mathematical and computational problem, see e.g., I. Jensen, “Enumeration of self-avoiding walks on the square lattice”, http://arxiv.org/abs/cond-mat/0404728, which lists 4190893020903935054619120005916 walks of length 71 steps.

The following code enumerates self-avoiding walks and also counts the number of cul-de-sacs and double cul-de-sacs, i.e., walks in which the walker finds itself in a dead end.

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

const int rMax = 50; // maximum length of walk
const int nMax = 2 * rMax + 1; // maximum number of lattice points in x, y
bool occupied[nMax][nMax]; // true if lattice site is occupied

inline int index(int xORy) { // translate x,y to array index
    return xORy + rMax;
}
```
bool isOccupied(int x, int y) { // check if site is occupied
    return occupied[index(y)][index(x)];
}

void occupy(int x, int y) { // mark the site occupied
    occupied[index(y)][index(x)] = true;
}

void clear(int x, int y) { // mark the site empty
    occupied[index(y)][index(x)] = false;
}

int nSteps; // desired number of steps in the walk
int steps; // number of steps so far
unsigned long walks; // number of walks so far with nSteps
double r2av; // accumulator for <r^2>
double r4av; // accumulator for <r^4>
int culDeSacs; // accumulator for cul-de-sacs
int doubleCulDeSacs; // accumulator for double cul-de-sacs

void initialize() { // empty sites and zero variables
    for (int y = -rMax; y <= rMax; y++)
for (int x = -rMax; x <= rMax; x++)
    clear(x, y);

r2av = r4av = 0;
walks = 0;
culDeSacs = doubleCulDeSacs = 0;
occupy(0, 0);
steps = 1;
}

int deadEnds(int x, int y, bool& isDoubleCulDeSac) {
    // count number of dead ends for this walk
    int deadEnds = 0;

    // is the start point surrounded by 4 occupied sites?
    if (isOccupied( 1, 0) && isOccupied(0, 1) &&
        isOccupied(-1, 0) && isOccupied(0, -1) ) ++deadEnds;

    // is the end point surrounded by 4 occupied sites?
    if (isOccupied(x + 1, y) && isOccupied(x, y + 1) &&
        isOccupied(x - 1, y) && isOccupied(x, y - 1) ) ++deadEnds;

    isDoubleCulDeSac = deadEnds == 2 && x*x + y*y != 1;
void visit(int x, int y) {
    if (isOccupied(x, y))
        return; // back up and continue
    occupy(x, y); // add this site to the walk
    ++steps;
    if (steps <= nSteps) { // recursively visit
        visit(x + 1, y); // East neighbor
        visit(x, y + 1); // North neighbor
        visit(x - 1, y); // West neighbor
        visit(x, y - 1); // South neighbor
        ++walks;
        double r2 = x*x + y*y; // start-to-end distance squared
        r2av += r2;
        r4av += r2 * r2;
        bool isDoubleCulDeSac;
        culDeSacs += dead Ends(x, y, isDoubleCulDeSac);
        if (isDoubleCulDeSac)
double CulDeSacs;
}

clear(x, y);       // remove this site from the walk
--steps;          // before backing up an continuing

int main() {

    cout << " Exact Enumeration of Self-Avoiding Walks on a Square Lattice\\n"
        << " ---------------------------------------------------------------------\\n"
        << " Enter maximum number of steps: ";
    int maxSteps;
    cin >> maxSteps;
    cout << " Steps  \(<r^2>\) Std. Dev. S.-A. Walks Cul-de-sacs "
        << " Double Cds  CPU secs" << endl;

double cpuTotal = 0;
unsigned long walksTotal = 0, cdsTotal = 0, dcdsTotal = 0;
for (nSteps = 1; nSteps <= maxSteps; nSteps++) {
    clock_t startTime = clock();
    initialize();
    visit(1, 0);
    r2av /= walks;
    cpuTotal += (clock() - startTime);
    walksTotal += walks;
    cdsTotal += CulDeSacs;
    dcdsTotal += doubleCulDeSacs;
}
r4av /= walks;
double stdDev = sqrt(r4av - r2av * r2av);
walksTotal += walks;
cdsTotal += culDeSacs;
dcdsTotal += doubleCulDeSacs;
clock_t endTime = clock();
double cpu = double(endTime - startTime) / CLOCKS_PER_SEC;
cpuTotal += cpu;
cout << right << setw(4) << nSteps << " " << left << setw(10) << r2av << " " << left << setw(10) << stdDev << " " << right << setw(10) << 4 * walks << " " << right << setw(10) << 4 * culDeSacs << " " << right << setw(10) << 4 * doubleCulDeSacs << " " << left << setw(8) << cpu << '
';

} 
cout << " " << setw(20) << "
 Totals " << right << setw(10) << 4 * walksTotal << " " << right << setw(10) << 4 * cdsTotal << " " << right << setw(10) << 4 * dcdsTotal << " " << left << setw(8) << cpuTotal << endl;
"}


The Reptation Method

The reptation model of polymer diffusion was developed by Pierre-Gilles 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.


Reptation Algorithm

The following algorithm implements the slithering snake model:

- 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

This Reptation Java Applet illustrates the algorithm.

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 in the Figure. If the two ends are not adjacent, the reptile can slither out of the configuration.

A reptile can slither out of the double dead end on the left but not out of the true double cul-de-sacs center and right. Double cul-de-sacs are rare.

**C++ Reptation Algorithm Code**

```
#include <cmath>
```

reptation.cpp
```cpp
#include <cstdlib>
#include <ctime>
#include <deque>
#include <iomanip>
#include <iostream>
#include <fstream>
#include <set>
using namespace std;

inline double std_rand()
{
    return rand() / (RAND_MAX + 1.0);
}

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

    // 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;
    }
};
```
Here we use a C++ deque object to store the current walk, and a C++ set object to store the set of occupied sites.

Because the Site object contains integer primitives, C++ uses the integer comparison to decide whether two sites are equal or not. In general, an object inserted in a standard container like a set must be Equality comparable and Less-than comparable. The latter must be defined for Site objects because the ordering of two points in a plane is not fixed by the values of the coordinates. The code will not compile unless operator< is explicitly defined.

```cpp
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();
}

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

const int // initial configuration choices
    STAIR = 1, // East-North random staircase
void createSnake(
    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
                std_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);
}
}

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;
alowed.push_back(s);
    }

break;
}

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

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

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

    Site head, neck, sNext;

    if (std_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 n_steps, n_walks;
cin >> n_steps;
cout << " Enter number of walks to generate: ";
cin >> n_walks;
cout << " Enter initial configuration 1 = stair, 2 = coil, 3 = line: ";
int config;
cin >> config;

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

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

    double r2sum = 0;
    double r4sum = 0;
    int success = 0;

clock_t startTime = clock();

createSnake(steps, config);

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

clock_t endTime = clock();

double r2av = r2sum / n_walks;
double stdDev = sqrt(r4sum / n_walks - r2av * r2av);
double successPercent = success / double(n_walks) * 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\";
Output of the Reptation Code

The Figure shows typical output of the reptation program. Note the dramatic growth of fluctuations with the length of the walk.
Self-Avoiding Walks using Reptation Algorithm

Number of Steps $n$

$<r^2>$

10^5 walks

10^7 walks