AEP 4380: Homework 9

Gregory Kaiser

November 22nd, 2019

1 Problem Background and Solution Overview

This assignment investigates an approach to analyzing protein folding from a simple two-dimensional model. Proteins are amino acids (biologically relevant molecules) linked by covalent bonds in a chain. When this chain is bent, van der Waals forces (among others) cause the chain to be attracted to itself. Through thermal fluctuations, and depending on the surrounding environment, proteins fold into complicated structures due to these attractions. This is how the protein folds into a functional structure.

Beginning with a simple straight line chain, one can simulate thermal fluctuations through the use of a random number generator (RNG) to allow the protein to change its shape. Successive randomizations can help the energy of the structure to fall (Monte Carlo calculations), and eventually level off at some minimization for a given structure. This is a simplification of how proteins fold in vivo.

The initial protein had an energy of 0 and looked like a vertical bar:

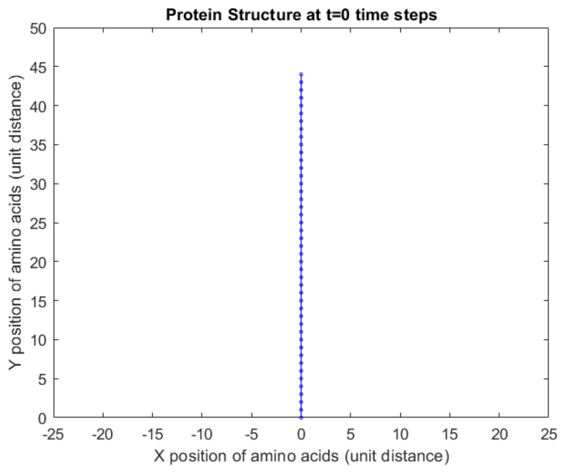


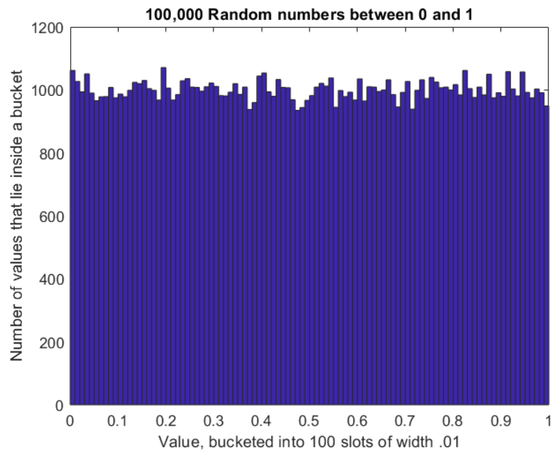
Figure 1: Protein initial condition.

In this two-dimensional model, amino acids lie on a grid with unit distance between each possible location. Covalent bonds between adjacent amino acids are locked at 1 unit distance, and so must lie at angles of 0, 90, and 180 degrees from a previous amino acid. Snapping to this grid allows random movements of each amino acid to be simplified greatly.

2 Solution Description

2.1 Random Number Generator (RNG)

The random number generator Ranq1.doub() is used to create a random distribution of 100,000 points, and a set of 10,000 random pairs of points.[1] This confirms that a call to this RNG generates a random double between 0 and 1 for use later.



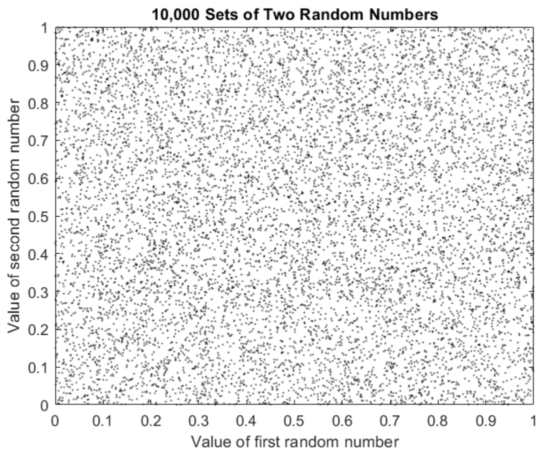
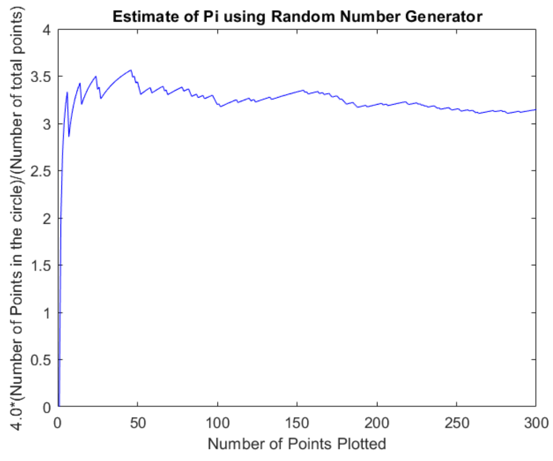


Figure 2: Left: A random distribution of 100,000 numbers between 0 and 1. Notice the average number of hits per bucket matches the bucket size of .01 (≈1000 per bucket). Right: 10,000 sets of 2 random numbers plotted as functions of one another. This is uniform as expected.

Interestingly, by counting the number of points that lie within a quarter of the unit circle centered on the origin (where the magnitude of the vector created by the two random numbers is less than 1), the value of π can be approximated over a large number of point samples.



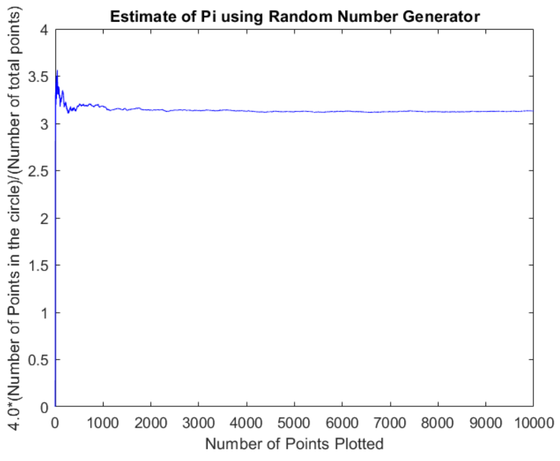


Figure 3: Through random sampling of the first quadrant, pi is approached using $4\frac{N_{incircle}}{N_{total}} \approx \pi$, and confirming the validity of the RNG. Left: a portion of the approximation to 300 points. Right: Full approximation to 10,000 points converges to around 3.1.

2.2 Protein Folding Algorithm

To initialize the starting protein, a two-dimensional arrayt of type *int* is used to store 45 amino acids, each with three attributes: x position, y position, and type of amino acid. There are 20 such possible amino acids.[2]

To calculate the energy of a given amino acid configuration, a 20x20 matrix of random numbers between $E_{min} = -7.0$ and $E_{max} = -2.0$ is initialized. For a given protein, energy total is calculated as:

$$E_{total} = \sum_{pairs i,j} E_{itype,jtype} \delta_{ij} \tag{1}$$

where *itype* and *jtype* are the type of amino acid at index i and j respectively.[3] This energy calculation is performed in a separate function, energy_calc, and iterates over all possible interactions with non-covalently bonded, but adjacent, amino acids. In other words, only amino acids which are on adjacent grid points and are not covalently bonded are counted in this calculation.

To generate a randomized new structure, a separate function, new_protein, takes in the old structure and modifies it with a valid movement of one amino acid. A RNG chooses a random amino acid in the sequence, chooses a possible direction for it to move (Northeast, Northwest, Southwest, or Southeast, diagonally from its current position), and checks to see if that move is conflicting with another amino acid location. The proposed new location of this amino acid is also checked with its neighbors to ensure the 1 unit distance maximum from each neighbor is not violated. The grid points not diagonal from the randomly chosen acid's original position are omitted because they would automatically violate this rule.

The energy of the new structure is calculated to check against the previous structure's energy. The value $\Delta E = E_{new} - E_{old}$ determines how much the energy changed from the last step due to this random change. If the new energy is the same or lower than the old one ($\Delta E \leq 0$), then this step is accepted as the new protein structure. If not, the new structure is accepted probabilistically based on the temperature of the system and k_B, Boltzmann's constant: A RNG creates a value between 0 and 1. If $e^{-\Delta E/k_B T} >$ the random number, then the new protein, however unfavorable, is accepted as a thermal fluctuation. Otherwise, the fluctuation is rejected. This is called the Metropolis algorithm.[3]

By performing the above many times, one is effectively simulating the fluctuations in a protein's structure, and allowing it to conform into a more closed shape over time.

3 Results and Interpretation

Using 10 million time steps and $k_B T = 1$, a randomly conformed protein was generated which matched expectations as it curled slowly from the ends towards the middle. Since the head and tail of the protein have more degrees of freedom than a center piece (and the new_protein algorithm is restricted to starting here), it makes sense that this is where thermal variations begin.

Energy as a function of time steps was plotted for about 500 spots during the simulation:

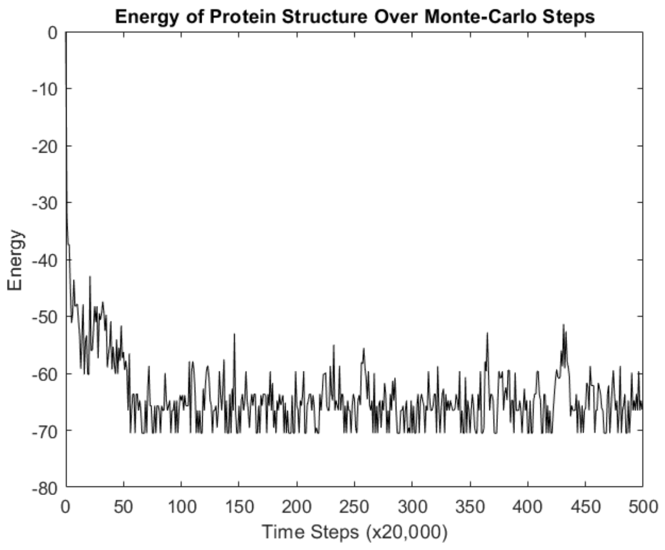


Figure 4: Energy of the protein as a function of Monte Carlo Steps

This approximately matches the expected exponential decay. Energy quickly drops as more folded configurations are found (all drastically more favorable than the striaght line), and then settles to some value from which it varies randomly.

The end-to-end distance of the protein structure was calculated with a modified distance calculation which uses doubles instead of integer distances.

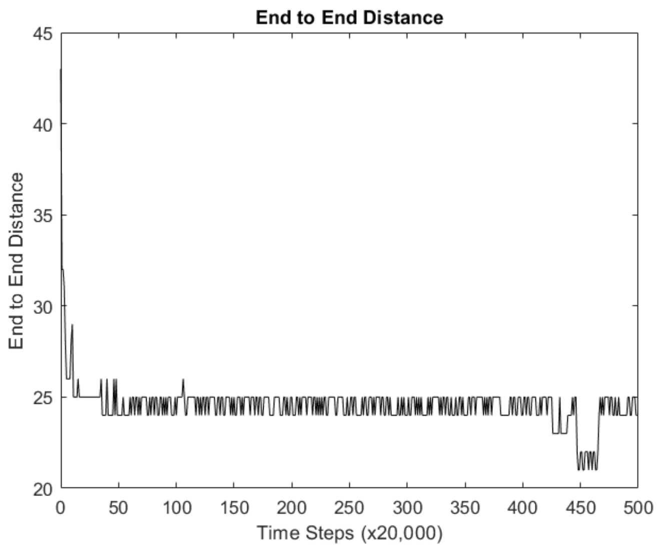


Figure 5: End-to-end distance of the protein as a function of Monte Carlo Steps

The actual protein structure is plotted at 10^4, 10^5, 10^6, and 10^7 time steps, to show the folding roughly as a function of time:

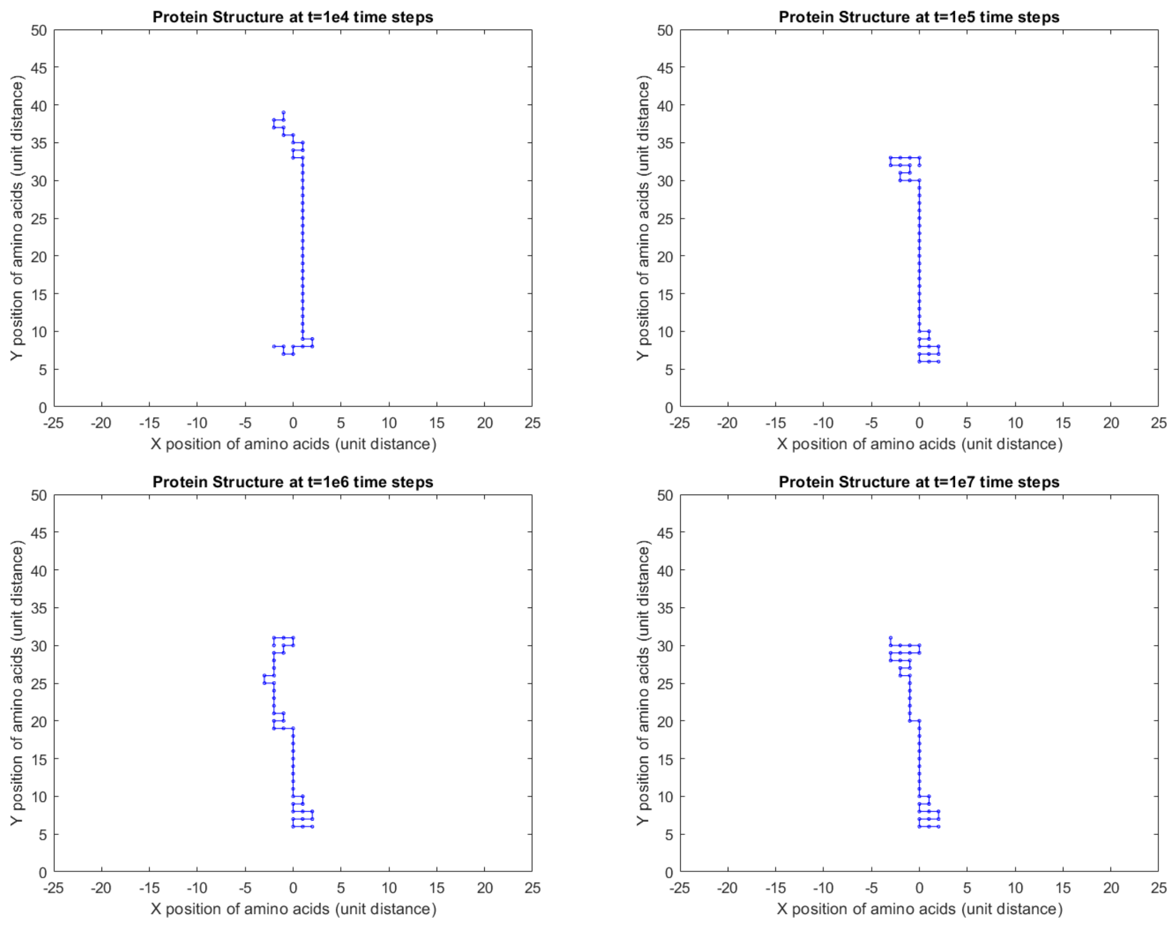


Figure 6: The protein structure plotted for time steps 1e4, 1e5, 1e6, 1e7 (final config).

This process is fascinating, since it allows something physical to be simulated statistically, and shows the importance and power of random-number-generators in general. To simulate Brownian motion, or statistical systems of any kind, these tools are necessary. Monte Carlo methods also have deep implications for optimization, since many iterations of a problem can be sampled and compared to find some ideal value by trial and error (an educated and directed guess and check).

3.1 Extra

Trying different seeds for the RNG yield drastically different results, and also a bit of fun. These differences are not only due to the variation in type of amino acid chosen along the chain, but also the randomized interaction energies.

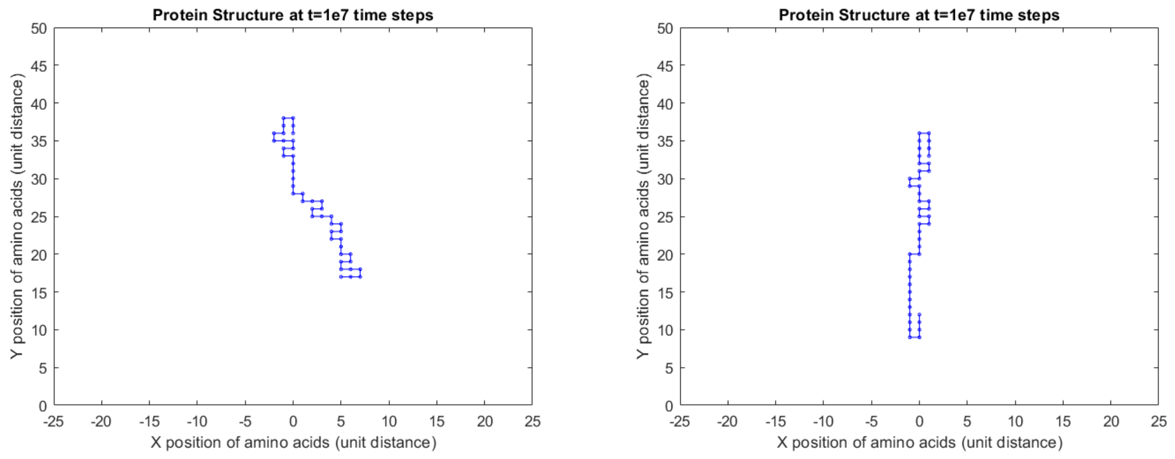


Figure 7: Two different seeds for the RNG yielf two completely different structures after 1e7 Monte Carlo calculation steps.

Using a constant seed, and averaging the energy over only the final 10,000 steps of the Monte Carlo calculation, a plot of average energy for a single amino acid string of length 45 as a function of k_BT shows how the protein has higher average energy at high temperature. It is more likely that structures with higher energy are accepted if k_BT is large, so this makes sense.

However, at low T, there isn't enough thermal variation to get the protein to find an optimal structure.

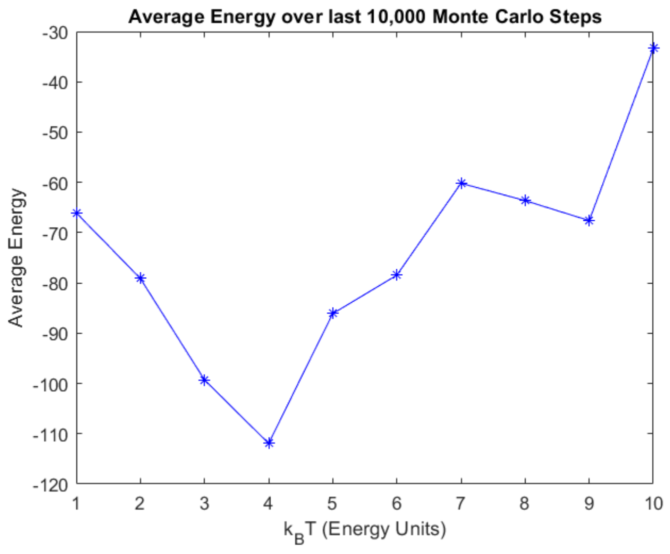
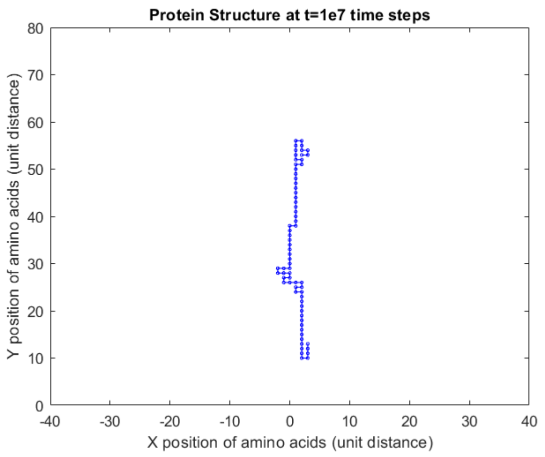


Figure 8: Rough plot of Average Energy over last 10,000 Monte Carlo steps versus k_BT from 10 to 1.

With a longer chain (70 amino acids), and with more Monte Carlo steps (100,000,000), not much more folding is observed But, a 60 amino acid chain after only 10 million steps is consistently more tangled.:



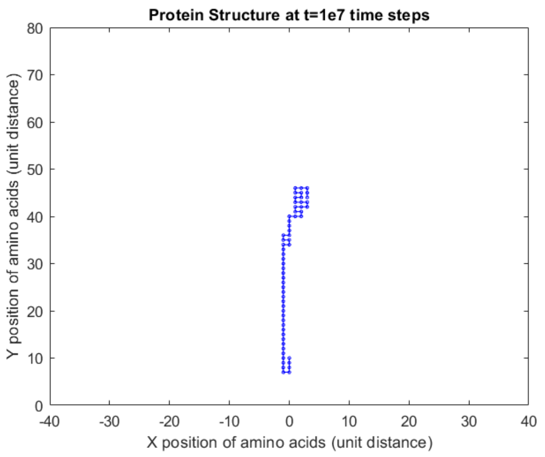


Figure 9: Left: Protein of 70 amino acids after 100 million Monte Carlo steps. Right: Protein of 60 amino acids after 10 million Monte Carlo steps.

This must mean that as one adds more amino acids, either the number of steps should be much higher, or $k_B T$ should be higher to introduce more thermal variation in the larger protein.

References

[1] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, *Numerical Recipes, The Art of Scientific Computing.* (3rd edit.), Cambridge Univ. Press, 2007, (ISBN 978-0-521-88068- 8, QA297 .N866 2007)

[2] Kirkland, Earl. *Array Class Objects in C/C++ for Vectors and Matrices* AEP 4380 Fall 2019. https://courses.cit.cornell.edu/aep4380/secure/arrayt.hpp

[3] Kirkland, Earl. *Monte Carlo Calculations* AEP 4380 Fall 2019 Assignment 9. https://courses.cit.cornell.edu/aep4380/secure/hw09f19.pdf

Source Code

```
/*  AEP 4380 Assignment #9
    Monte Carlo Calculations

    Run on Windows core i7 with gcc version 8.2.0 (MinGW.org GCC-8.2.0-3)

    Gregory Kaiser November 22nd 2019

*/
#include <cstdlib>
//#include <cstdis>
#include <cmath>
//#include <ctime>
#include <iostream>
#include <fstream>
#include <iomanip>
//Numerical Recipes type definitions for use of rendom number  generator Press et. al.
#include "nr3.h"
//Numerical Recipes RNG Press et. al.
#include "ran.h"
//#define ARRAYT_BOUNDS_CHECK
//from class website -
//Kirkland, E: https://courses.cit.cornell.edu/aep4380/secure/arrayt.hpp
#include "arrayt.hpp"//for vectors, matrices, ease of use

using namespace std;

//function definitions
double energy_calc(arrayt<double> &, arrayt<int> &);
void new_protein(arrayt<int> &);
void print_protein(arrayt<int> &);
int distance(int , int , arrayt<int> &);
int distance_doub(int , int , arrayt<int> &);
int distance_pos(int, int , int , int );

//RNG seed
Ullong iseed = 222774874;//time(NULL);//new seed on every run
Ranq1 my_rand = Ranq1(iseed);
int main(){

    ofstream fp; //output file for random numbers
    fp.precision(8);

    fp.open("hw9_1.dat");
    if(fp.fail()){
        cout<<"cannotopenfile"<<endl;
        return(EXIT_SUCCESS);
    }

    ofstream fp2; //output file for convergence to pi
    fp2.precision(8);

    fp2.open("hw9_2.dat");
    if(fp2.fail()){
        cout<<"cannotopenfile"<<endl;
        return(EXIT_SUCCESS);
    }
```

```
ofstream fp3; //output file for energy
fp3.precision(8);

fp3.open("hw9_3.dat");
if(fp3.fail()){
    cout<<"cannotopenfile"<<endl;
    return(EXIT_SUCCESS);
}
//HW Assignment

//-----TESTING RANQ1 AND FINDING PI------------
int i, j, k, num_rand = 10000, num_circ=0;
//num_rand = 100000;//for the full 100,000 distribution
double pi_est = 0;

arrayt<double> rand_1(num_rand);//100000 random number array
arrayt<double> rand_2(num_rand);//another random number array

for(i=0;i<num_rand;i++){
    rand_1(i) = my_rand.doub();//fill with random numbers between 0 and 1
    rand_2(i) = my_rand.doub();//fill with random numbers between 0 and 1
    fp<<rand_1(i)<<setw(15)<<rand_2(i)<<setw(15)<<i<<endl;
    if((rand_1(i)*rand_1(i)+rand_2(i)*rand_2(i))<=1){
        //this spot is in a circle radius 1
        num_circ++;
    }
    fp2<<4.0*num_circ/(i+1)<<setw(15)<<(i+1)<<endl;
}
//-----END TESTING RANQ1----------------

double E_min = -7.0, E_max = -2.0, energy = 0;
int length_prot = 70, xpos_init=0, ypos_init=0;
arrayt<double> E_amino(20,20);
arrayt<int> protein(3,length_prot);

//initialize the interaction energy matrix
for(i=0;i<20;i++){
    for(j=i;j<20;j++){
        E_amino(i,j) = my_rand.doub()*(E_max-E_min)+E_min;
        E_amino(j,i) = E_amino(i,j);
        //cout<<E_amino(i,j)<<setw(15)<<E_amino(j,i)<<endl;
    }
}

//fill initial protein
for(j=0;j<length_prot;j++){
    //xposition
    protein(0,j) = xpos_init;
    //cout<<protein(0,j)<<endl;
    //yposition
    protein(1,j) = ypos_init;
    //type
    protein(2,j) = (int) floor(my_rand.doub()*20);
    //cout<<protein(2,j)<<endl;
    //increment to initialize a striaght line protein
    ypos_init++;
}
```

```
    //print_protein(protein);
    energy = energy_calc(E_amino, protein);
    cout<<"E_0: "<<energy<<endl;

    //-----------Iterate by calculating new energies------------
    int Nsteps = 100000000, energy_i=0;
    double delta_energy=0, energy_prev=energy, kbT = 1.0, e_avg=0;
    double rand_e=0, end2end=0;
    //create a temporary structure to test for energy before accepting
    arrayt<int> temp(3,length_prot);

    for(i=0;i<Nsteps;i++){
        temp = protein; //dummy temp to test new structure
        //generate a random new structure and check its energy
        new_protein(temp);
        energy = energy_calc(E_amino, temp);//calc the new energy
        delta_energy = energy-energy_prev;//difference from prev

        if(delta_energy<=0){//good new protein-->keep it
            protein = temp;
            energy_prev=energy;
        }
        else{//this protein has higher energy, accept probabilistically
            rand_e = my_rand.doub();
            if((exp(-delta_energy/kbT))>rand_e){//accept the move
                protein = temp;
                energy_prev=energy;
            }
            else{//reject the change
                //protein is unchanged
                //previous energy is unchanged
            }
        }

        if(i%10000==0){//marker for progress
            cout<<i<<endl;
        }
        if(i==1e6){//marker for plotting specific time steps
           // print_protein(protein);
        }
        if(i%20000==0){//plot 500 points for the energy and end to end distance
            end2end = distance_doub(0,length_prot-1,protein);
            fp3<<end2end<<setw(15)<<energy<<setw(15)<<energy_i<<endl;
            energy_i++;
        }
        if(i>(Nsteps-10000)){//average over last 10000 terms
            e_avg+=energy;
        }

    }
    print_protein(protein);//final configuration
    cout<<"E_"<<i<<": "<<energy<<endl;//final energy
    cout<<e_avg/10000<<endl;

    fp.close();
    fp2.close();
    fp3.close();
    return(EXIT_SUCCESS);
} //end main
```

```
//helper to print the protein to the console for debugging
//and to print to the file for interpretation in MATLAB
void print_protein(arrayt<int> &protein){
    ofstream fp_prot; //output file for protein

    fp_prot.open("hw9_prot.dat");
    if(fp_prot.fail()){
        cout<<"cannotopenfile"<<endl;
    }

    int j, length = protein.n2();
    for(j=0;j<length;j++){
        //cout<<j<<setw(15)<<protein(0,j)<<setw(15)<<
        //    protein(1,j)<<setw(15)<<protein(2,j)<<endl;
        fp_prot<<j<<setw(15)<<protein(0,j)<<setw(15)<<
            protein(1,j)<<setw(15)<<protein(2,j)<<endl;
    }

    fp_prot.close();
}

//modifies old protein into a new allowed configuration
//does not end until new configuration is chosen
void new_protein(arrayt<int> &old_protein){
    int i,j, length = old_protein.n2();

    int ix, iy, itype;//positions of the ith animo acid
    int n1x, n1y, n2x, n2y, dist, dist2; //covalently bonded neighbor positions
    //indicators of new found protein and validity of change
    int chosen = 0, valid_move=1, neighbors_ok=1;
    int num_iter=0;
    //choose a random amino acid in the sequence
    int rand_acid;
    int rand_acid_x;
    int rand_acid_y;
    //chose a random new place to put the randomly chosen amino acid
    int rand_move;
    //store the proposed move positions
    int new_pos_x;
    int new_pos_y;

    while(!chosen){
    //if(1){
        //choose a random amino acid in the sequence
        rand_acid = (int) floor(my_rand.doub()*(length)); //its index

        //if length=45, rand_acid is a number between 0 and 44
        rand_acid_x = old_protein(0,rand_acid); //its xpos
        rand_acid_y = old_protein(1,rand_acid); //its ypos
        //chose a random new place to put the randomly chosen amino acid
        rand_move = (int) floor(my_rand.doub()*4)+1;
        //1,2,3,4, corresponding to NE, NW, SW, SE movement

        //store the proposed move positions
        switch(rand_move){
            case 1://move northeast, delta x and y is +1
                new_pos_x = rand_acid_x+1;
                new_pos_y = rand_acid_y+1;
                break;
            case 2://move northwest, delta x is -1 and y is +1
                new_pos_x = rand_acid_x-1;
                new_pos_y = rand_acid_y+1;
                break;
            case 3://move southwest, delta x and y is -1
                new_pos_x = rand_acid_x-1;
                new_pos_y = rand_acid_y-1;
                break;
            case 4://move southeast, delta x is +1 and y is -1
                new_pos_x = rand_acid_x+1;
                new_pos_y = rand_acid_y-1;
                break;
            default://random number generated incorrectly, throw error
                cout<<"Invalid Random Move Chosen"<<endl;
                break;
        }

        //check spatial conflict of proposed move and all AAs in sequence
        i=0; valid_move=1;
        while(i<length&&valid_move){
            if(i!=rand_acid){//this is some acid other than the randomly chosen one
                //it’s positions:
                ix = old_protein(0,i);
                iy = old_protein(1,i);
                if((new_pos_x==ix)&&(new_pos_y==iy)){//this means there is a conflict
                    valid_move=0;//ends checking for conflicts
                }
            }//end other acid check
            i++;
        }//end looking at all the amino acids for spatial conflict

        neighbors_ok=1;//reset neighbor check
        if(valid_move){//no spatial conflicts, check neighbor distances
            if(rand_acid==0){//head
                n1x = old_protein(0,rand_acid+1);
                n1y = old_protein(1,rand_acid+1);
                dist = distance_pos(new_pos_x, new_pos_y, n1x, n1y);
                if(dist!=1){//this AA isn’t the right distance for the proposed change
                    neighbors_ok=0;
                }
            }
            else if(rand_acid==length-1){//tail
                n2x = old_protein(0,rand_acid-1);
                n2y = old_protein(1,rand_acid-1);
                dist = distance_pos(new_pos_x, new_pos_y, n2x, n2y);
                if(dist!=1){//this AA isn’t the right distance for the proposed change
                    neighbors_ok=0;
                }
            }
            else{//middle
                n2x = old_protein(0,rand_acid-1);
                n2y = old_protein(1,rand_acid-1);
                n1x = old_protein(0,rand_acid+1);
                n1y = old_protein(1,rand_acid+1);
                dist2 = distance_pos(new_pos_x, new_pos_y, n2x, n2y);
                dist = distance_pos(new_pos_x, new_pos_y, n1x, n1y);
                if(dist!=1||dist2!=1){
                    //this AA isn’t the right distance from either AA for the proposed change
                    neighbors_ok=0;
                }
            }
        }

        if(valid_move&&neighbors_ok){
            //this means that the move checks out, both spatially and
            //with distance to neighbors
            //modify old protein and don't try another random change
            chosen = 1;
            old_protein(0, rand_acid) = new_pos_x; //modify old x val
            old_protein(1, rand_acid) = new_pos_y; //modify old y val
        }
        else{//continue to try random changes without changing the old protein
            chosen = 0;
        }
        num_iter++;
    }

}

//calculates the interaction energy of a given structure with a set of
//interaction energies
double energy_calc(arrayt<double> &Energies, arrayt<int> &structure){
    int i,j, ix, iy, itype, size = structure.n2();
    int n1x, n1y, n2x, n2y,  dist; //covalently bonded neighbor positions
    int jx, jy, jtype;//possible neighbor positions
    double energy = 0;
    //cycle through all amino acids
    for(i=0;i<size;i++){
        //store ith amino acid
        itype = structure(2,i);
        for(j=i+2;j<size;j++){
            //since j<i+1 have been checked against all j already
            //and j==i+1 is a covalent bond for sure
                dist = distance(i,j,structure);
                if(dist==1){//valid neighbor, non covalent
                    //neighbor positions to check
                    jtype = structure(2,j);
                    energy+=Energies(itype,jtype);
                }//end valid neighbor
        }//end j
    }//end i
    return energy;
}//end calc energy

//calculate the distance between two amino acids
int distance(int index1, int index2, arrayt<int> &structure){
    int posx1, posx2, posy1, posy2;
    posx1 = structure(0,index1);
    posy1 = structure(1,index1);
    posx2 = structure(0,index2);
    posy2 = structure(1,index2);
    return abs(posx1-posx2)+abs(posy1-posy2);
}

//calculate the distance between two amino acids
//more accurate for the distance plot as a double
int distance_doub(int index1, int index2, arrayt<int> &structure){
    double posx1, posx2, posy1, posy2;
    posx1 = (double) structure(0,index1);
    posy1 = (double) structure(1,index1);
    posx2 = (double) structure(0,index2);
    posy2 = (double) structure(1,index2);
    return sqrt(((posx1-posx2)*(posx1-posx2))+((posy1-posy2)*(posy1-posy2)));
}

//calculate the distance between two (x, y) sets.
int distance_pos(int posx1, int posy1, int posx2, int posy2){
    return abs(posx1-posx2)+abs(posy1-posy2);
}
```