Monte Carlo Simulation of Protein Folding

Yangrui Hu[1]

[1] *Department of Physics, Brown University, Providence, Rhode Island 02912, USA*

(Dated: May 8, 2018)

Protein folding is a hot topic both in biology and biophysics. There are many scientists working on the simulation of protein folding system. Although Molecular Dynamics method is a good technique to probe this problem, the calculation is huge when the size of the system becomes large. In this case, Monte Carlo method can give us a rough result quickly and help us understand this system. In this paper, I will use Monte Carlo method to simulate proteins consisting of 15, 30, and 100 amino acids under different temperatures. The energy and end-to-end distance are two key quantities to be studied. By Markov Chain Monte Carlo simulation, I find that the protein tends to stay in the highly folded and stable state in low temperature. There is a phase transition for energy and end-to-end distance as changing temperature and the transition temperature is independent of the number of amino acids. When temperature is high enough, the energy and end-to-end distance approach to the result of self-avoiding random walk system, which indicates that the effect of thermal fluctuation dominates.

INTRODUCTION

Proteins are one of the most important components of cells and organisms. Proteins perform a large number of functions in biochemical processes such as energy storage, transfer, inter-cellular communication, and work as catalysts in biochemical reactions. As a kind of macromolecules, a protein is composed of many chains of amino acids of which there are twenty kinds known to exist in nature. Different combinations of amino acids form thousands of kinds of protein molecules. The primary structures of proteins are their linear amino-acid sequences, and their specific structures determine their functions and properties.[2] When a protein works in a biological activity, its structure is the high-order structure, generally speaking, as known as the folded state. The first step to study how a protein works or to design a protein with a new kind of function is to figure out its folded structure. According to Anfinsen's research[1], heating proteins or putting them in some chemical environments can cause protein denaturation and unfolding. However, if we put them back into the original environment, the polypeptide chain will fold back into its native state in just a second. Anfinsen concluded that the resulting three-dimensional structure (native state) is determined by the amino acid sequence or primary structure[1]. This conclusion shows the essence of protein folding process.

To build a computable model, forces or energies which influence the folded structure need to be considered. There are four key factors taken into account[11]: the Van der Waals force between non-neighboring amino acids; the hydrogen bonding; interactions between amino acids and water molecules or other chemical molecules in the solution; and the thermal fluctuation. The Van der Waals force can be described by Lennard-Jones potential: $U(r) = 4\varepsilon[(\frac{\sigma}{r})^{12} - (\frac{\sigma}{r})^{6}]$, which is attractive when the distance r is not too large. The hydrogen bonding is also attractive and leads to the folded structure. As for interactions between amino acids and water molecules, some amino acids are hydrophilic or water-loving, and others are hydrophobic or water-fearing. Hydrophilic amino acids tend to unfold to increase the probability of touching water molecules. On the other hand, hydrophobic amino acids tend to fold. In general, corresponding energies of forces mentioned above are in the order of k_BT. So, the thermal fluctuation leading to the increase of the disorder of the system has to be considered.

In this paper, I will simulate the process of the protein folding with the Monte Carlo method and monitor two key quantities: the energy of the system and the end-to-end distance which is the distance between the first amino acid and the last one. Moreover, I will study the effect of temperature, interaction coefficient, and the number of amino acids. In the second section, I will briefly introduce the Metropolis algorithm and simulated annealing algorithm which I will use in my simulations. In the third section, I will talk about the computational model for this problem and the simulation algorithm. In the fourth and fifth sections, I will present my main simulation results, conclusions and discussions.

BACKGROUND AND THEORY

The study of protein folding includes the prediction of protein stability, kinetics, and structure. In the late 1980s and early 1990s, Joseph Bryngelson and Peter Wolynes established the formulation of an energy landscape theory of proteins and they introduced the principle of minimal frustration[5]. The minimal frustration principle means that nature has chosen amino acid sequences so that the folded state of the protein is very stable. Moreover, the undesired interactions between amino acids along the folding pathway are reduced, and consequently the folding process is very fast. These evolutionarily selected sequences result in that proteins are

thought to have globally funneled energy landscapes, which makes the protein fold to the native state through a large number of pathways[9]. Although nature has this minimal frustration principle, some degree of frustration remains up and can be observed as the protein staying in local minima in the energy landscape. This theory is supported by some experimental results and simulation results[5] and helps us to predict and design the protein structure.

There are some available computational methods for protein folding is summarized by a recent review[4]. Molecular Dynamics method is an important technique to study the protein folding process[12]. Nevertheless, the calculation is huge when the size of the system becomes large. In this case, Monte Carlo method can give us a rough result quickly and help us understand this system.

For a protein system, we can simply assume the protein is an amino acid chain consisting of N amino acids and located on a 2D grid. Each amino acid is on a grid point. This chain forms a self-avoiding walk(SAW) pattern, which means the chain can not touch itself. Then we can define the energy of this system which will be discussed in the later section. Metropolis algorithm and simulated annealing method will be applied in my simulations.

Metropolis-Hastings Algorithm

Metropolis-Hastings algorithm[7] is a kind of Markov Chain Monte Carlo algorithm. According to the master equation, we can get the detailed balance equation[10]:

$$\frac{P(x \to x')}{P(x' \to x)} = \frac{P(x')}{P(x)}, \tag{1}$$

where $P(x \to x')$ is the probability of that state x evolves to state x'. $P(x' \to x)$ is the probability of that state x' evolves to state x. $P(x)$ is the probability of state x and $P(x')$ is the probability of state x'.

Then we can divide the evolution probability into two parts:

$$P(x \to x') = g(x \to x')P_{convert}(x \to x'), \tag{2}$$

where $g(x \to x')$ is the suggestion probability, and $P_{convert}(x \to x')$ is the conversion probability. Then we get:

$$\frac{P_{convert}(x \to x')}{P_{convert}(x' \to x)} = \frac{P(x')g(x' \to x)}{P(x)g(x \to x')}. \tag{3}$$

Finally we can choose appropriate conversion probability that satisfies detailed balance condition:

$$P_{convert}(x \to x') = \min(1, \frac{P(x')g(x' \to x)}{P(x)g(x \to x')}). \tag{4}$$

So Metropolis-Hastings algorithm can be summarized as following:

1. Choose an initial state and set $t = 0$.
2. Choose a random state x for iteration and generate a new state x' randomly.
3. Calculate the acceptance probability.
4. Accept or reject the new state based on the acceptance probability.
5. Set $t = t + 1$ and continue to iterate.

For the protein system, this system is a kind of canonical ensemble. So for any state α, the probability of this state is $\exp(-E_\alpha/k_BT)$. Then when we have two states: old state and new state, their relative probability is $r = \exp(-\Delta E/k_BT)$. In other words, when we consider to move the system from the old state to the new state, the acceptance probability of the new state is $\exp(-E_\alpha/k_BT)$.

Simulated Annealing Method

For some systems like Spin Glass and protein, there are many possible metastable states[6]. When we do the simulation, we can not just use general algorithm discussed before. We have to take care not to let the system fall into the metastable state. Simulated annealing method is a way to solve this problem. The general idea was raised by Kirkpatrick[8]: the metastable states are maybe at the local minimum in the phase space. However, the stable state should be at the global minimum. In the low temperature, k_BT is small and the acceptance probability $\exp(-\Delta E/k_BT) \ll 1$ when $\Delta E > 0$, which means that it is not easy for the system to escape from the local minimum. If we increase temperature of the system immediately, the acceptance probability also increases dramatically, and the system can escape from the local minimum. After that, we can reduce temperature gradually, and it is easier for the system to find the global minimum. This process is similar as the annealing process, so we call it as simulated annealing method.

METHOD

In order to simplify the problem, I assume the protein is an amino acid chain. Suppose an amino acid chain with N amino acids is located on a 2D grid, and each amino acid is on a grid point, forming a SAW pattern[6].

Neighboring amino acids are linked by covalent bonds. The amino acid sequence can be represented by a $N \times 1$ array $A[N]$, where $A[i] = 1, 2, 3, ..., 20$, corresponding to twenty kinds of amino acids. The interaction energy between non-covalent amino acids m and n is $J_{A(m),A(n)}$. So the total interaction energy (I will use "energy" to represent it) of the protein is

$$E = \sum_{<m,n>} J_{A(m),A(n)}, \tag{5}$$

where $< m, n >$ means a pair of non-covalent neighboring amino acids. Figure 1 shows examples of possible states for an amino acid chain. According to the definition of the energy, the first three states are degenerate and the energy is zero because there is no non-covalent neighboring amino acid pair. The number of pairs that contribute to the energy in the last state is 1. The interaction coefficient $J_{A(m),A(n)}$ will be set as negative which is reasonable since that net forces between amino acids are usually attractive, so the potential should be negative. Given the initial condition, we can use Metropolis method to do the simulation and the detailed iteration algorithm is shown as following[13]:

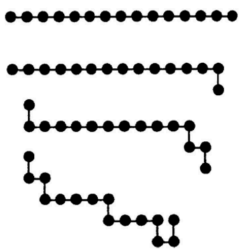

FIG. 1. Four examples of possible states for an amino acid chain[6]

- Choose a node of the chain randomly and its coordinate is (x_0, y_0).
- Choose one of its next nearest neighbors (x_n, y_n) randomly.
- If moving the node to the new position (x_n, y_n) doesn't change the length of the chain, calculate the change in energy ΔE after the movement.
- Apply Metropolis algorithm to decide whether to accept this movement: if $\Delta E < 0$, accept; else the acceptance probability is $e^{-\beta \Delta E}$, where $\beta = \frac{1}{k_B T}$, k_B is Boltzmann constant, and T is temperature of the system.

Figure 2 shows an example for an iteration. First we can choose a node, for example the point with number 5. It has four next nearest neighbors: number 1, 3, 7, and 9. Apparently, only number 9 is possible to be the new position that we can accept. After a long time, when the system reaches the thermal equilibrium, we can find the final state, study statistical properties of this system, and discuss effects of temperature, the interaction coefficient, and the length of the chain N. When we simulate the protein chain in low temperature, as I mentioned before, we have to use simulated annealing method: set high temperature initially and then decrease the temperature gradually.

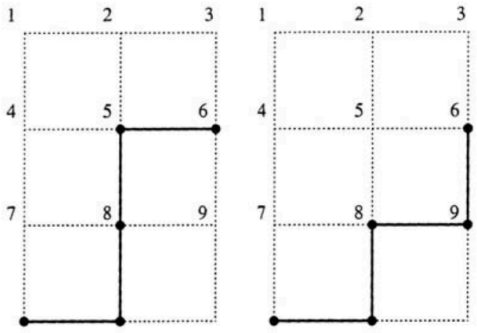

FIG. 2. An example for an iteration[6]

SIMULATION RESULTS

In this section, I will present my main simulation results. First of all, I should test my code. I monitor the evolution of the energy and end-to-end distance of proteins with 100 amino acids when $T = 1$ both with simulated annealing method and without using it. Then I study the effect of temperature, interaction coefficient J, and the number of amino acids N.

Thermal Equilibrium

In order to test my code, the evolution of the energy and end-to-end distance of proteins has been recorded. The initial condition of the protein with N amino acids is $(x_i, y_i) = (i, 0)$, where $i = 0, 1, 2, ..., N-1$. Repeat the iteration process mentioned in the Method section a large number of times, and only a part of steps can generate a new configuration. First, we can record the energy and the end-to-end distance of the system in each of these steps. Figure 3 shows the evolution of the energy and end-to-end distance (length) of the protein when $N = 100$ and $T = 1$. The same system has been simulated by 10 times. In order to simplify the calculation, I set the Boltzmann constant k_B as 1, so temperature is in the unit of Boltzmann constant. The unit of time is the Monte Carlo step. Because temperature is low, simulated annealing method is applied in this simulation.

From Figure 3 we can clearly find that the energy of the system decreases continuously. Figure 4 shows the simulation results for the same system without using simu-

lated annealing method. Compare Figure 3 and Figure 4 and we can see that simulated annealing method does work for low-temperature case. The energy at the thermal equilibrium when $T = 1$ with simulated annealing method is approximately -120, which is much smaller than the energy without simulated annealing method (~ -50). The final length with simulated annealing method is around 40, which is also much smaller than the final length without simulated annealing method (~ 70).

Figure 3 and Figure 4 also shows there are big fluctuations between different trials. Thus, simulating the system for many times is needed to improve statistical accuracy.

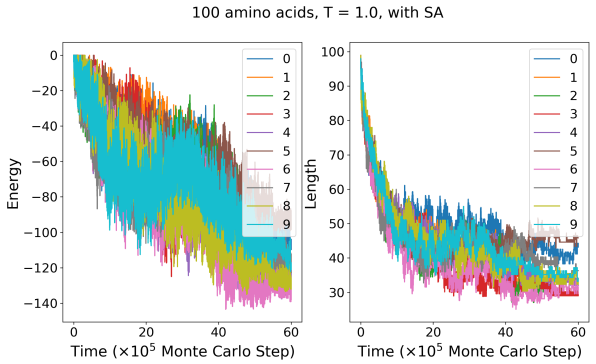

FIG. 3. Evolution of the energy and end-to-end distance (length) of the protein consisting of 100 amino acids when T = 1. Simulated annealing method is applied in this simulation.

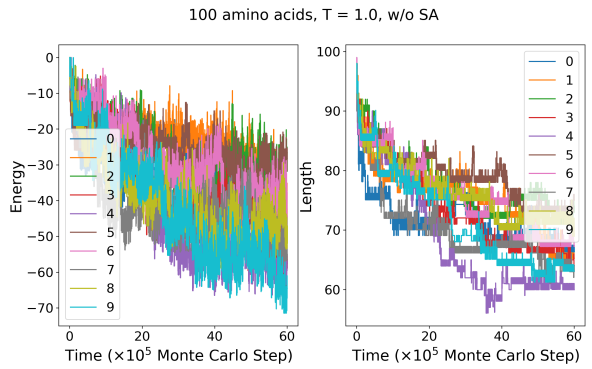

FIG. 4. Evolution of the energy and end-to-end distance (length) of the protein consisting of 100 amino acids when T = 1.

Effect of Temperature and J

After the system staying in the thermal equilibrium, we can calculate the time-average energy and end-to-end distance in different temperatures. Figure 5 shows simulation results for the protein with 15 amino acids. From Figure 5 we can see that trends of the evolution for energy and length are similar. Length is a quantity which can describe the extent of folding. Highly folded proteins have more pairs of non-covalent neighboring amino acids, so their energies should be lower given the definition. In other words, qualitatively speaking, energy is proportional to the length and this is the reason that the evolution of length has similar trend as energy.

The role of interaction coefficient J is nothing but to rescale the temperature. Based on the acceptance probability $\exp(-\Delta E/k_B T) = \exp(-J\Delta n/T)$, where Δn is the change in number of pairs that contribute to the energy, the key term is the ratio J/T. Consequently, if we choose large J, we can get more details in high temperature; similarly, if we set small J, we can get more details in low temperature. Figure 5 also confirms this statement: trends of both evolutions of energy and length are similar with different J.

In Figure 5, I set two kinds of J: uniform J and $J_{A(m),A(n)}$ dependent on the type of amino acids. For the latter case, I set $J_{A(m),A(n)}$ is a random number in the range of $[-2, -4]$. It's clear that $J = -3$ case and $J \in [-2, -4]$ case have almost the same results. Thus, J will not influence key behaviors of proteins.

As for the effect of temperature, in low temperature, the extent of folding tends to be high since the length tends to be small. As raising temperature, there is a phase transition. Further study about this phase transition will be discussed in the later section. When temperature is high enough, the energy and length stop to increase and approach to the certain value. Later study will show that the certain value is result of self-avoiding random walk.

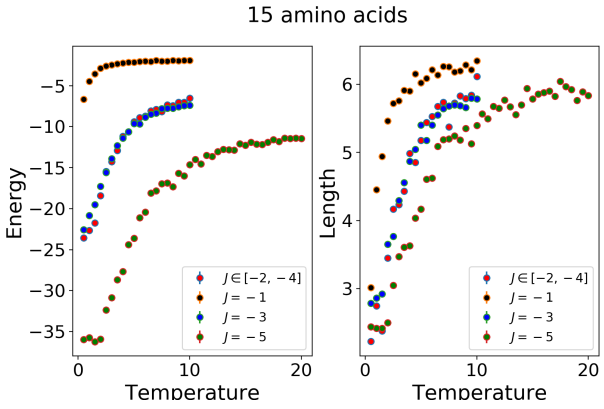

FIG. 5. The relationship between temperature and the energy (left) and end-to-end distance (right) of the protein consisting of 15 amino acids with different interaction coefficient J.

Effect of N

Figure 6 shows simulation results with 15, 30, 100 amino acids. For different N, evolutions of energy have similar trends, so rules discussed before are general and independent of N. There is a phase transition for all N like ferromagnetic phase transition. Figure 6 also tells us that N doesn't change the transition temperature.

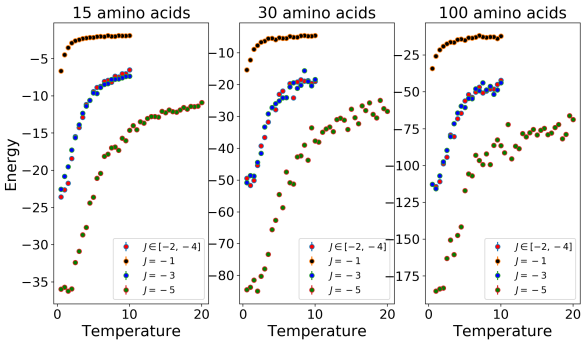

FIG. 6. The relationship between temperature and the energy of the protein consisting of 15, 30, 100 amino acids with different interaction coefficient J.

The Certain Value: Self-Avoiding Random Walk

During the study of the effect of temperature, I find the energy and length stop to increase and approach to the certain values when temperature is high enough. According to the acceptance probability $\exp(-\Delta E/k_B T)$, when temperature is high enough, $\exp(-\Delta E/k_B T)$ will approach to 1. The difference between self-avoiding random walk system and protein system is that acceptance probability of the former one is always 1. Therefore, theoretically, when temperature high enough, results of protein system should approach to that of self-avoiding system.

In order to confirm my assumption, I simulate the self-avoiding random walk system with 15, 30, 100 amino acids and compare their results with protein systems as Figure 7. The interaction coefficient J is -1 because I want to focus on the high temperature part. Figure 7 clearly shows that the certain value is the result of self-avoiding random walk system.

Phase Transition

During the study of the effect of temperature, I find there is a phase transition for energy and length as changing temperature. Critical exponent is a significant parameter for the phase transition phenomenon[3] and I

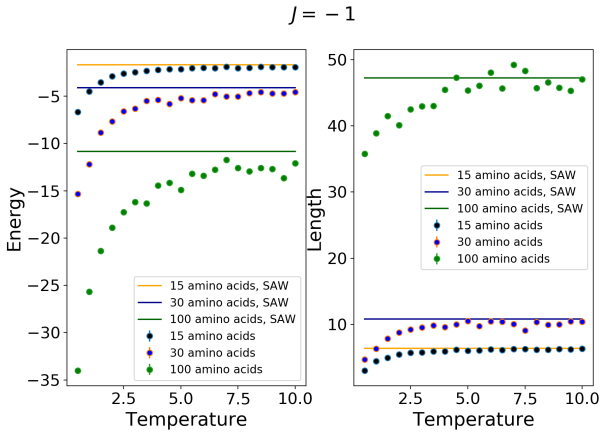

FIG. 7. The comparison of Self-Avoiding Random Walk simulation and protein folding simulation for chains with 15, 30, 100 amino acids. The interaction coefficient J is -1.

am going to calculate the critical exponent for 15 amino acid case. The definition of the critical exponent is

$$k = \lim_{\tau \to 0} \frac{\log\left(f(|\tau|)\right)}{\log\left(|\tau|\right)}, \tag{6}$$

where $\tau = (T - T_c)/T_c$ is the reduced temperature. This result can also be written as power law: $f(\tau) \propto \tau^k$. Figure 8 shows the fitting result using power fitting function and the critical exponent for energy is 0.61 and for length is 0.15. However, this result is not accurate because the fitting function is nonlinear. According to the definition of the critical exponent, the relationship between $\log\left(f(|\tau|)\right)$ and $\log\left(|\tau|\right)$ is linear around $\tau = 0$. Figure 9 shows the linear fitting results, and the critical exponent for energy is 0.7 and for length is 0.15. For the transition temperature, I use minimum chi square method to find its optimized value: $T_c = 2.0$.

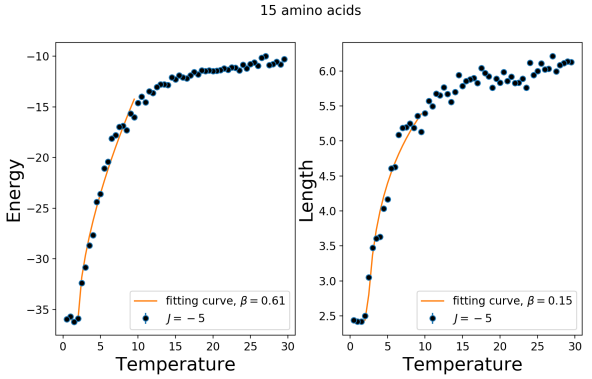

FIG. 8. The fitting results of critical exponents for chains with 15 amino acids. The interaction coefficient J is -5.

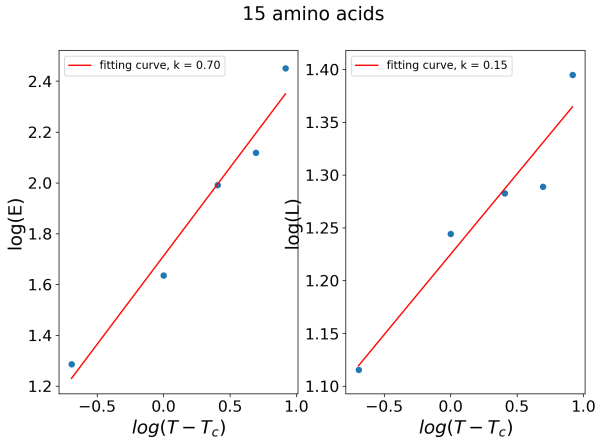

FIG. 9. The linear fitting results of $\ln(E)$, $\ln(L)$ vs $\ln(T-T_c)$. The interaction coefficient J is -5.

CONCLUSION & DISCUSSION

From my simulations, there are three main conclusions can be obtained:

1. **The extent of folding tends to be high in low temperature.** Length or end-to-end distance is a quantity which can describe the extent of folding. Small end-to-end distance means large extent of folding, so according to simulation results, the end-to-end distance of the protein is short and the protein tends to become highly folded in low temperature. Highly folded state has lower energy and is more stable. In low temperature, the effect of interaction potential dominates, and the minimal frustration principle[5] will make the acquisition of the folded state.

2. **Phase transition is observed in the protein system and the transition temperature is independent of N.** Phase transition phenomena mean that the protein will fold or unfold suddenly when temperature goes through the critical temperature. This is consistent with experiments[1]. Experiments found that heating or putting proteins into some chemical environments can cause protein denaturation or unfolding, and proteins can fold back into its native state in just a second if we put them back into the original environment.

3. **The protein system approaches to self-avoiding random walk system when temperature is high enough.** The acceptance probability in protein system is high in high temperature, so the effect of the thermal fluctuation is dominant, and the behavior of the system is like self-avoiding random walk system. In nature, when temperature is high enough, the amino acid chain will break, so the system becomes a random system.

Overall, Metropolis algorithm and simulated annealing method can simulate the protein folding dynamics well, although the folding funnel landscape allows the protein to fold to the native state through any of a large number of pathways and intermediates, rather than being restricted to a single mechanism[5]. The minimal frustration principle makes proteins have a mechanism like simulated annealing method. On the other hand, some degree of frustration remains up and nature allows the protein to stay in metastable states[5]. As for phase transition phenomena, further study is needed to explain it and give theoretical prediction of critical exponents. The study of the protein folding mechanism is needed. In my work, I have only just touched on this interesting topic.

In summary, my work confirms that Monte Carlo simulations can help us understand the protein folding problem quickly. Three significant conclusions can be obtained by Monte Carlo simulations. Although the model in my work is simple, it can indeed reveal some key ideas.

I would like to thank Brown University to support my work. I would also like to thank Prof. Kemp Plumb for his help and guidance to this topic.

[1] Christian B. Anfinsen. The formation and stabilization of protein structure. *Biochemical Journal.*, 128(4):737–749, 1972.
[2] Christian B. Anfinsen. Principles that govern the folding of protein chains. *Science*, 181(4096):223–230, 1973.
[3] Stephen Blundell. Magnetism in condensed matter, 2003.
[4] Mario Compiani and Emidio Capriotti. Computational and theoretical methods for protein folding. *Biochemistry*, 52(48):8601–8624, 2013.
[5] Bryngelson Joseph D., Onuchic Jos Nelson, Socci Nicholas D., and Wolynes Peter G. Funnels, pathways, and the energy landscape of protein folding: A synthesis. *Proteins: Structure, Function, and Bioinformatics*, 21(3):167–195.
[6] Zejun Ding. *Lecture Notes for Computational Physics*. manuscript.
[7] WK Hastings. Monte-Carlo Sampling Methods Using Markov Chains and Their Applications. *Biometrika*, 57(1):97–&, 1970.
[8] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.
[9] P E Leopold, M Montal, and J N Onuchic. Protein folding funnels: a kinetic approach to the sequence-structure relationship. *Proceedings of the National Academy of Sciences*, 89(18):8721–8725, 1992.
[10] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21(6):1087–1092, 1953.
[11] C N Pace, B A Shirley, M McNutt, and K Gajiwala.

Forces contributing to the conformational stability of proteins. *The FASEB Journal*, 10(1):75–83, 1996. PMID: 8566551.

[12] Bruno Rizzuti and Valerie Daggett. Using simulations to provide the framework for experimental protein folding studies. *Archives of Biochemistry and Biophysics*, 531(1):128 – 135, 2013. Protein Folding and Stability.

[13] Jay Wang. *Computational Modeling and Visualization of Physical Systems with Python*. John Wiley & Sons.

A Appendix: Code for simulating protein folding

Listing 1: Original Script for simulating protein folding

```
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
from numba import jit
import random

matplotlib.rcParams.update({'font.size': 15})

@jit(nopython=True)
def step_2D_jit():
    d = np.random.randint(1,5)
    if d == 1:
        x_s, y_s = 1, 1
    if d == 2:
        x_s, y_s = 1, -1
    if d == 3:
        x_s, y_s = -1, 1
    if d == 4:
        x_s, y_s = -1, -1
    return x_s, y_s

@jit(nopython=True)
def neighbor(x, y, x_test, y_test):
    if (x-x_test)**2+(y-y_test)**2 < 1.1:
        return 1
    if (x-x_test)**2+(y-y_test)**2 > 1.1:
        return 0

@jit(nopython=True)
def Calculate_delta_Energy_jit(N_aminoacids, R, J, A, index, xnew, ynew, temp):

    k = index
    x_new = xnew
    y_new = ynew
    R_new = np.zeros((N_aminoacids, 2))
    R_before = np.zeros((N_aminoacids, 2))
    T = temp
    beta = 1.0/T

    for i in range(N_aminoacids):
        R_new[i,0] = R[i,0]
        R_new[i,1] = R[i,1]
        R_before[i,0] = R[i,0]
        R_before[i,1] = R[i,1]
        if i == k :
            R_new[i,0] = x_new
            R_new[i,1] = y_new


    energy_before = 0
```

```
    energy_new = 0
    distance_before = 0
    distance_new = 0

    for i in range(N_aminoacids):
        for j in range(i):
            if i-j > 1.1:
                distance_before = (R_before[i,0]-R_before[j,0])**2 +
                (R_before[i,1]-R_before[j,1])**2
                distance_new = (R_new[i,0]-R_new[j,0])**2 + (R_new[i,1]-
                    R_new[j,1])**2
                if distance_before < 1.1:
                    energy_before = energy_before - J[A[i],A[j]]
                if distance_new < 1.1:
                    energy_new = energy_new - J[A[i],A[j]]

    delta_energy = energy_new - energy_before
    if delta_energy <= 0:
        R[k,0] = x_new
        R[k,1] = y_new
    elif delta_energy > 0:
        p = random.random()
        if p < np.exp(-beta*delta_energy):
            R[k,0] = x_new
            R[k,1] = y_new
    return R

@jit(nopython=True)
def ProteinFolding_jit(N_aminoacids, J, A, Temp = 10, Nsteps = 1000000):

    R = np.zeros((N_aminoacids, 2))
    # Initialize the protein chain, R is the position array
    for i in range(N_aminoacids):
        R[i,0] = i # x_i
        R[i,1] = 0 # y_i

    FreeEnergy = []
    length = []
    step = []

    for i in range(Nsteps):
        # choose k-th amino acid randomly and make the movement
        k = np.random.randint(0, N_aminoacids)
        x = R[k,0]
        y = R[k,1]
        x_s, y_s = step_2D_jit()
        x_new = x + x_s
        y_new = y + y_s

        #judge whether the walking is self-avoiding
        SAW = 1
        for j in range(N_aminoacids):
            if (R[j,0]-x_new)**2+(R[j,1]-y_new)**2 < 0.1:
```

```
                SAW = 0
    if SAW == 0 :
        continue

    #judge whether the length is not changed
    if k == 0:
        x_next = R[k+1,0]
        y_next = R[k+1,1]
        nbor = neighbor(x_new, y_new, x_next, y_next)
        if nbor == 1:
            R = Calculate_delta_Energy_jit(N_aminoacids, R, J, A, k,
                x_new, y_new, Temp)
        if nbor == 0:
            continue

    if k == N_aminoacids-1:
        x_pre = R[k-1,0]
        y_pre = R[k-1,1]
        nbor = neighbor(x_new, y_new, x_pre, y_pre)
        if nbor == 1:
            R = Calculate_delta_Energy_jit(N_aminoacids, R, J, A, k,
                x_new, y_new, Temp)
        if nbor == 0:
                continue

    if 0 < k < N_aminoacids-1:
        x_next = R[k+1,0]
        y_next = R[k+1,1]
        x_pre = R[k-1,0]
        y_pre = R[k-1,1]
        nbor_next = neighbor(x_new, y_new, x_next, y_next)
        nbor_pre = neighbor(x_new, y_new, x_pre, y_pre)
        if nbor_next == 1 and nbor_pre == 1:
            R = Calculate_delta_Energy_jit(N_aminoacids, R, J, A, k,
                x_new, y_new, Temp)
        if nbor_pre == 0:
            continue
        if nbor_next == 0:
            continue

    # calculate the free energy, r square
    F = 0
    for j in range(N_aminoacids):
        for l in range(j):
            if j-l > 1.1:
                distance = (R[j,0]-R[l,0])**2 + (R[j,1]-R[l,1])**2
                if distance < 1.1:
                    F = F - J[A[j],A[l]]

    r2 = (R[N_aminoacids-1,0]-R[0,0])**2 + (R[N_aminoacids-1,1]-R[0,1])
    **2
    step.append(i+1)
    FreeEnergy.append(F)
```

```
            length.append(np.sqrt(r2))

        return step, FreeEnergy, length, R

@jit(nopython=True)
def ProteinFolding_low_temp_jit(N_aminoacids, J, A, Nsteps = 1000000):

    R = np.zeros((N_aminoacids, 2))
    # Initialize the protein chain, R is the position array
    for i in range(N_aminoacids):
        R[i,0] = i # x_i
        R[i,1] = 0 # y_i

    FreeEnergy = []
    length = []
    step = []

    for i in range(Nsteps):

        if 0 <= i < 2000000:
            temp = 10.0
        if 2000000 <= i < 2500000:
            temp = 9.5
        if 2500000 <= i < 3000000:
            temp = 9.0
        if 3000000 <= i < 3500000:
            temp = 8.5
        if 3500000 <= i < 4000000:
            temp = 8.0
        if 4000000 <= i < 4500000:
            temp = 7.5
        if 4500000 <= i < 5000000:
            temp = 7.0
        if 5000000 <= i < 5500000:
            temp = 6.5
        if 5500000 <= i < 6000000:
            temp = 6.0
        if 6000000 <= i < 6500000:
            temp = 5.5
        if 6500000 <= i < 7000000:
            temp = 5.0
        if 7000000 <= i < 7500000:
            temp = 4.5
        if 7500000 <= i < 8000000:
            temp = 4.0
        if 8000000 <= i < 8500000:
            temp = 3.5
        if 8500000 <= i < 9000000:
            temp = 3.0
        if 9000000 <= i < 9500000:
            temp = 2.5
        if 9500000 <= i < 10000000:
            temp = 2.0
```

```
        if 10000000 <= i < 10500000:
            temp = 1.5
        if 10500000 <= i < 11000000:
            temp = 1.0
        if 11000000 <= i < 11500000:
            temp = 0.5

        # choose k-th amino acid randomly and make the movement
        k = np.random.randint(0, N_aminoacids)
        x = R[k,0]
        y = R[k,1]
        x_s, y_s = step_2D_jit()
        x_new = x + x_s
        y_new = y + y_s

        #judge whether the walking is self-avoiding
        SAW = 1
        for j in range(N_aminoacids):
            if (R[j,0]-x_new)**2+(R[j,1]-y_new)**2 < 0.1:
                SAW = 0
        if SAW == 0 :
            continue

        #judge whether the length is not changed
        if k == 0:
            x_next = R[k+1,0]
            y_next = R[k+1,1]
            nbor = neighbor(x_new, y_new, x_next, y_next)
            if nbor == 1:
                R = Calculate_delta_Energy_jit(N_aminoacids, R, J, A, k,
                    x_new, y_new, temp)
            if nbor == 0:
                continue

        if k == N_aminoacids-1:
            x_pre = R[k-1,0]
            y_pre = R[k-1,1]
            nbor = neighbor(x_new, y_new, x_pre, y_pre)
            if nbor == 1:
                R = Calculate_delta_Energy_jit(N_aminoacids, R, J, A, k,
                    x_new, y_new, temp)
            if nbor == 0:
                continue

        if 0 < k < N_aminoacids-1:
            x_next = R[k+1,0]
            y_next = R[k+1,1]
            x_pre = R[k-1,0]
            y_pre = R[k-1,1]
            nbor_next = neighbor(x_new, y_new, x_next, y_next)
            nbor_pre = neighbor(x_new, y_new, x_pre, y_pre)
            if nbor_next == 1 and nbor_pre == 1:
                R = Calculate_delta_Energy_jit(N_aminoacids, R, J, A, k,
```

```
                        x_new, y_new, temp)
                if nbor_pre == 0:
                    continue
                if nbor_next == 0:
                    continue

            # calculate the free energy, r square
            F = 0
            for j in range(N_aminoacids):
                for l in range(j):
                    if j-l > 1.1:
                        distance = (R[j,0]-R[l,0])**2 + (R[j,1]-R[l,1])**2
                        if distance < 1.1:
                            F = F - J[A[j],A[l]]

            r2 = (R[N_aminoacids-1,0]-R[0,0])**2 + (R[N_aminoacids-1,1]-R[0,1])
            **2
            step.append(i+1)
            FreeEnergy.append(F)
            length.append(np.sqrt(r2))

    return temp, step, FreeEnergy, length, R


def initialize(N_aminoacids):
    A = np.zeros(N_aminoacids, int)
    J = np.zeros((20,20))
    # Set the sequence of the protein chain A[N_aminoacids]
    for i in range(N_aminoacids):
        A[i] = np.random.randint(0,20)

    # Set the interaction energy matrix J
    for i in range(20):
        for j in range(i):
            J[i,j] = 2 * random.random() + 2
            J[j,i] = J[i,j]
        J[i,i] = random.random()

    return A, J

def initialize_Uniform_J(N_aminoacids):
    A = np.zeros(N_aminoacids, int)
    J = np.zeros((20,20))+ 5
    for i in range(N_aminoacids):
        A[i] = np.random.randint(0,20)
    return A, J

def plot_folding_structure(N_aminoacids, R):

    for i in range(N_aminoacids-1):
        x1, y1 = R[i,0], R[i,1]
        x2, y2 = R[i+1,0], R[i+1,1]
        x = [x1, x2]
```

```
            y = [y1, y2]
            plt.plot(x,y,'r')
        for i in range(N_aminoacids):
            plt.scatter(R[i,0], R[i,1], c ='k')
    plt.xlabel('x', fontsize=20)
    plt.ylabel('y', fontsize=20)
    plt.title('Final Structure (T = 10, 1000000 steps)', fontsize=20)
    plt.show()


def equilibrium(N_aminoacids):

    Temp = 1
    A, J = initialize(N_aminoacids)

    for n in range(10):
        step, FreeEnergy, length, R = ProteinFolding_jit(N_aminoacids, J
            , A, Temp, 6000000)
        #t, step, FreeEnergy, length, R = ProteinFolding_low_temp_jit
        (N_aminoacids, J, A, 6000000)
        file = open('full_equilibrium_%d_T%.1f-beforeSA-%d.txt'
            %(N_aminoacids, Temp, n), 'w')
        for i in range(len(step)):
            file.write(str(step[i]) + " " + str(FreeEnergy[i]) + " " +
             str(length[i]) + "\n")

    #plot_folding_structure(N_aminoacids, R)


def phase_transition(N_aminoacids):

    A, J = initialize_Uniform_J(N_aminoacids)

    #high temperature simulation
    N_steps = 2000000
    ave_int = 1800000
    temp_int = 30
    temp_final = 20
    dtemp = 0.5
    temp_step = int((temp_int - temp_final)/dtemp) + 1
    temp_array = np.linspace(temp_int, temp_final, temp_step, endpoint=True)

    for t in temp_array:

        file = open('UniJ5_equilibrium_%d_T%.1f.txt'%(N_aminoacids, t), 'w')

        for n in range(10):

            step, FreeEnergy, L, R = ProteinFolding_jit(N_aminoacids, J,
             A, t, N_steps)

            for i in range(len(step)):
                if step[i] < ave_int:
```

```
                    continue
                if step[i] > ave_int:
                    file.write(str(step[i]) + " " + str(FreeEnergy[i]) +
                     " " + str(L[i]) + "\n")

    #low temperature simulation
    #low_temp_array = np.linspace(9.5, 0.5, 19, endpoint=True)

    #for t in low_temp_array:

    #    if t > 1.5:
    #        continue

    #    Nsteps = 2500000 + int(500000 * ( 9.5 - t) * 2)
    #    ave_int_low = Nsteps - 200000

    #    file = open('UniJ5_equilibrium_%d_T%.1f(1).txt'%(N_aminoacids, t), 'w')

    #    for n in range(20):

    #        temp, step, FreeEnergy, L, R = ProteinFolding_low_temp_jit
    (N_aminoacids, J, A, Nsteps)
    #        #print(temp)
    #        for i in range(len(step)):
    #            if step[i] < ave_int_low:
    #                continue
    #            if step[i] > ave_int_low:
    #                file.write(str(step[i]) + " " + str(FreeEnergy[i])
    + " " + str(L[i]) + "\n")


@jit(nopython=True)
def SAW_ProteinFolding_jit(N_aminoacids, J, A, Nsteps = 1000000):

    R = np.zeros((N_aminoacids, 2))
    # Initialize the protein chain, R is the position array
    for i in range(N_aminoacids):
        R[i,0] = i # x_i
        R[i,1] = 0 # y_i

    FreeEnergy = []
    length = []
    step = []

    for i in range(Nsteps):
        # choose k-th amino acid randomly and make the movement
        k = np.random.randint(0, N_aminoacids)
        x = R[k,0]
        y = R[k,1]
        x_s, y_s = step_2D_jit()
        x_new = x + x_s
        y_new = y + y_s
```

```
        #judge whether the walking is self-avoiding
        SAW = 1
        for j in range(N_aminoacids):
             if (R[j,0]-x_new)**2+(R[j,1]-y_new)**2 < 0.1:
                  SAW = 0
        if SAW == 0 :
             continue

        #judge whether the length is not changed
        if k == 0:
             x_next = R[k+1,0]
             y_next = R[k+1,1]
             nbor = neighbor(x_new, y_new, x_next, y_next)
             if nbor == 1:
                  R[k,0], R[k,1] = x_new, y_new
             if nbor == 0:
                  continue

        if k == N_aminoacids-1:
             x_pre = R[k-1,0]
             y_pre = R[k-1,1]
             nbor = neighbor(x_new, y_new, x_pre, y_pre)
             if nbor == 1:
                  R[k,0], R[k,1] = x_new, y_new
             if nbor == 0:
                       continue

        if 0 < k < N_aminoacids-1:
             x_next = R[k+1,0]
             y_next = R[k+1,1]
             x_pre = R[k-1,0]
             y_pre = R[k-1,1]
             nbor_next = neighbor(x_new, y_new, x_next, y_next)
             nbor_pre = neighbor(x_new, y_new, x_pre, y_pre)
             if nbor_next == 1 and nbor_pre == 1:
                  R[k,0], R[k,1] = x_new, y_new
             if nbor_pre == 0:
                  continue
             if nbor_next == 0:
                  continue

        # calculate the free energy, r square
        F = 0
        for j in range(N_aminoacids):
             for l in range(j):
                  if j-l > 1.1:
                       distance = (R[j,0]-R[l,0])**2 + (R[j,1]-R[l,1])**2
                       if distance < 1.1:
                            F = F - J[A[j],A[l]]

        r2 = (R[N_aminoacids-1,0]-R[0,0])**2 + (R[N_aminoacids-1,1]
             -R[0,1])**2
        step.append(i+1)
```

```
            FreeEnergy.append(F)
            length.append(np.sqrt(r2))

    return step, FreeEnergy, length


def SAW(N_aminoacids):

    #to confirm the assumption is correct
    N_steps = 2000000
    ave_int = 1800000
    A, J = initialize_Uniform_J(N_aminoacids)

    file = open('SAW_UniJ1_%d.txt'%N_aminoacids, 'w')

    for n in range(100):

        step, FreeEnergy, L = SAW_ProteinFolding_jit(N_aminoacids,
         J, A, N_steps)
        for i in range(len(step)):
                if step[i] < ave_int:
                    continue
                if step[i] > ave_int:
                    file.write(str(step[i]) + " " + str(FreeEnergy[i])
                        + " " + str(L[i]) + "\n")

def main():

    phase_transition(15)
    #equilibrium(100)
    #SAW(100)


if __name__ == '__main__':
    main()
```

B Appendix: Code for analyzing the simulation data

Listing 2: Original Script for analyzing simulation data

```
import numpy as np
import matplotlib.pyplot as plt

def equilibrium_mean_SD(fname):
    a = np.loadtxt(fname)
    E = a[:,1]
    L = a[:,2]
    std_E = np.std(E)
    std_L = np.std(L)
    se_E = std_E/np.sqrt(len(E))
    se_L = std_L/np.sqrt(len(L))
    return np.mean(E), np.mean(L), std_E, se_E, std_L, se_L

def phase_data(N_aminoacids = 15):
    temp_int = 30
    temp_final = 0.5
    dtemp = 0.5
    temp_step = int((temp_int - temp_final)/dtemp) + 1
    temp_array = np.linspace(temp_int, temp_final, temp_step, endpoint=True)
    file = open('UniJ5_phase_%d.txt'%N_aminoacids, 'w')
    for t in temp_array:
        fname = "UniJ5_equilibrium_%d_T%.1f.txt"%(N_aminoacids, t)
        mean_E, mean_L, std_E, se_E, std_L, se_L = equilibrium_mean_SD(fname)
        file.write(str(t) + " " + str(mean_E) + " " + str(se_E) + " " +
            str(std_E) + " " + str(mean_L) +  " " + str(se_L)+ " " + str(std_L)
            + "\n")

def SAW_data(N_aminoacids = 15):
    temp_int = 10
    temp_final = 0.5
    dtemp = 0.5
    temp_step = int((temp_int - temp_final)/dtemp) + 1
    temp_array = np.linspace(temp_int, temp_final, temp_step, endpoint=True)

    fname = 'SAW_UniJ1_%d.txt'%N_aminoacids
    a = np.loadtxt(fname)
    E = a[:,1]
    L = a[:,2]
    mean_E = np.mean(E)
    mean_L = np.mean(L)
    std_E = np.std(E)
    std_L = np.std(L)
    se_E = std_E/np.sqrt(len(E))
    se_L = std_L/np.sqrt(len(L))

    file = open('SAW_UniJ1_phase_%d.txt'%N_aminoacids, 'w')
    for t in temp_array:
        file.write(str(t) + " " + str(mean_E) + " " + str(se_E) + " " + str(std_E)
            + " " + str(mean_L) +  " " + str(se_L)+ " " + str(std_L) + "\n")

```

```
def main():
     phase_data(N_aminoacids = 15)
     #SAW_data(100)

if __name__ == '__main__':
     main()
```

C Appendix: Code for plotting the evolution of the energy and length

Listing 3: Original Script for plotting the evolution of the energy and length

```
import numpy as np
import matplotlib
import matplotlib.pyplot as plt

matplotlib.rcParams.update({'font.size': 17})

def plot_equilibrium(fname, N_aminoacids, Temp):
    a = np.loadtxt(fname)
    step0 = a[:,0]
    E = a[:,1]
    L = a[:,2]

    step = step0[:]*0.00001

    fig = plt.figure()
    ax1 = fig.add_subplot(121)
    ax2 = fig.add_subplot(122)

    ax1.plot(step, E, label='%d amino acids, T = %.1f, w/o SA'%(N_aminoacids, Temp))
    ax2.plot(step, L, label='%d amino acids, T = %.1f, w/o SA'%(N_aminoacids, Temp))
    ax1.set_xlabel("Time ($\\times 10^5$ Monte Carlo Step)", fontsize=20)
    ax1.set_ylabel("Energy", fontsize=20)
    ax1.legend(fontsize=15)
    ax2.set_xlabel("Time ($\\times 10^5$ Monte Carlo Step)", fontsize=20)
    ax2.set_ylabel("Length", fontsize=20)
    ax2.legend(fontsize=15)
    plt.show()

def plot_SA_compare(N_aminoacids, Temp):
    fig = plt.figure()
    ax1 = fig.add_subplot(121)
    ax2 = fig.add_subplot(122)

    for n in range(10):
        fname = 'full_equilibrium_%d_T%.1f-beforeSA-%d.txt'%(N_aminoacids, Temp, n)
        a = np.loadtxt(fname)
        step0 = a[:,0]
        step = step0[:]*0.00001
        E = a[:,1]
        L = a[:,2]
        ax1.plot(step, E, label = '%d'%n)
        ax2.plot(step, L, label = '%d'%n)

    ax1.set_xlabel("Time ($\\times 10^5$ Monte Carlo Step)", fontsize=20)
    ax1.set_ylabel("Energy", fontsize=20)
    ax1.legend()
    ax2.set_xlabel("Time ($\\times 10^5$ Monte Carlo Step)", fontsize=20)
    ax2.set_ylabel("Length", fontsize=20)
```

```
    ax2.legend()
    plt.suptitle('%d amino acids, T = %.1f, w/o SA'%(N_aminoacids, Temp), fontsize=20)
    plt.show()


def main():
    N_aminoacids = 100
    Temp = 1.0
    #fname = 'full_equilibrium_%d_T%.1f-beforeSA-%d.txt'%(N_aminoacids, Temp, n)
    #plot_equilibrium(fname, N_aminoacids, Temp)
    plot_SA_compare(N_aminoacids, Temp)

if __name__ == '__main__':
    main()
```

D Appendix: Code for plotting the phase diagram

Listing 4: Original Script for plotting and fitting the phase diagram

```
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit

matplotlib.rcParams.update({'font.size': 12})
#plt.subplots_adjust(wspace = 0.5)

def func_1(x, a, b):
     return a*x+b

def analysis_phase_transition():
     a = np.loadtxt('UniJ5_phase_15_full.txt')

     temp_array = a[:,0]
     E = a[:,1]
     se_E = a[:,2]
     L = a[:,4]
     se_L = a[:,5]
     file = open('fitJ5_15.txt', 'w')
     for t in range(11):
          Tc = 1.5 + t * 0.1
          yE = []
          yL = []
          delta_E = []
          delta_L = []
          x = []
          for i in range(len(temp_array)):
               if Tc < temp_array[i] <= 5:
                    yE.append(np.log(abs(E[i]+37)))
                    delta_E.append(se_E[i]/abs(E[i]+37))
                    yL.append(np.log(abs(L[i])))
                    delta_L.append(se_E[i]/L[i])
                    x.append(np.log(abs(temp_array[i]-Tc)))

          poptE, pcovE = curve_fit(func_1, x, yE)
          poptL, pcovL = curve_fit(func_1, x, yL)
          chi2_E = 0
          chi2_L = 0
          for i in range(len(x)):
               chi2_E += (yE[i]-poptE[0]*x[i]-poptE[1])**2
               chi2_L += (yL[i]-poptL[0]*x[i]-poptL[1])**2

          file.write(str(Tc) + " " + str(poptE[0]) + " " + str(chi2_E) + " "
               + str(poptL[0]) +  " " + str(chi2_L) + "\n")

def plot_fit_phasetransition():
     fig = plt.figure()
     ax1 = fig.add_subplot(121)
     ax2 = fig.add_subplot(122)
```

```
    b = np.loadtxt('fitJ5_15.txt')
    Tc = b[:,0]
    kE = b[:,1]
    chi2E = b[:,2]
    kL = b[:,3]
    chi2L = b[:,4]
    ax1.plot(Tc, kE, label='critical exponent')
    ax1.set_ylabel("k", fontsize=12)
    ax1.legend(fontsize=12)
    ax3 = ax1.twinx()
    ax3.plot(Tc,chi2E, 'red', label = '$\chi^2$')
    ax3.set_xlabel("$T_c$", fontsize=12)
    ax3.set_ylabel('$\chi^2$',fontsize=12)
    ax3.legend(fontsize=12)
    ax1.set_title('Energy')

    ax2.plot(Tc, kL, label='critical exponent')
    ax2.set_ylabel("k", fontsize=12)
    ax2.legend(fontsize=12)
    ax4 = ax2.twinx()
    ax4.plot(Tc, chi2L, 'red', label='$\chi^2$')
    ax4.set_xlabel("$T_c$", fontsize=12)
    ax4.set_ylabel('$\chi^2$',fontsize=12)
    ax4.legend(fontsize=12)
    ax2.set_title('Length')
    plt.suptitle('15 amino acids, $J = -5$')
    plt.show()

def funcE(x,a,r):
    Tc = 2.0
    if x > Tc:
        return a*(x/Tc-1)**r-36
    if x <= Tc:
        return -36

def fit_funcE(x,a,r):
    f = np.zeros(len(x))
    for i in range(len(f)):
        f[i] = funcE(x[i],a,r)
    return f

def funcL(x,a,r):
    Tc = 2.0
    if x > Tc:
        return a*(x/Tc-1)**r-2.44
    if x <= Tc:
        return 2.44

def fit_funcL(x,a,r):
    f = np.zeros(len(x))
    for i in range(len(f)):
        f[i] = funcL(x[i],a,r)
```

```
    return f

def plot_15_phase():

    fig = plt.figure()
    ax1 = fig.add_subplot(121)
    ax2 = fig.add_subplot(122)

    #a = np.loadtxt('phase_15.txt')
    #a1 = np.loadtxt('UniJ1_phase_15.txt')
    #a1_SAW = np.loadtxt('SAW_UniJ1_phase_15.txt')
    #a3 = np.loadtxt('UniJ3_phase_15.txt')
    #a3_SAW = np.loadtxt('SAW_UniJ3_phase_15.txt')
    a5 = np.loadtxt('UniJ5_phase_15_full.txt')
    #a5_SAW = np.loadtxt('SAW_UniJ5_phase_15.txt')

    #temp_array = a[:,0]
    #E = a[:,1]
    #se_E = a[:,2]
    #std_E = a[:,3]
    #L = a[:,4]
    #se_L = a[:,5]
    #std_L = a[:,6]

    #temp_array1 = a1[:,0]
    #E1 = a1[:,1]
    #se_E1 = a1[:,2]
    #std_E1 = a1[:,3]
    #L1 = a1[:,4]
    #se_L1 = a1[:,5]
    #std_L1 = a1[:,6]

    #temp_array1_SAW = a1_SAW[:,0]
    #E1_SAW = a1_SAW[:,1]
    #se_E1_SAW = a1_SAW[:,2]
    #std_E1_SAW = a1_SAW[:,3]
    #L1_SAW = a1_SAW[:,4]
    #se_L1_SAW = a1_SAW[:,5]
    #std_L1_SAW = a1_SAW[:,6]

    #temp_array3 = a3[:,0]
    #E3 = a3[:,1]
    #se_E3 = a3[:,2]
    #std_E3 = a3[:,3]
    #L3 = a3[:,4]
    #se_L3 = a3[:,5]
    #std_L3 = a3[:,6]

    #temp_array3_SAW = a3_SAW[:,0]
    #E3_SAW = a3_SAW[:,1]
    #se_E3_SAW = a3_SAW[:,2]
    #std_E3_SAW = a3_SAW[:,3]
    #L3_SAW = a3_SAW[:,4]
```

```
#se_L3_SAW = a3_SAW[:,5]
#std_L3_SAW = a3_SAW[:,6]

temp_array5 = a5[:,0]
E5 = a5[:,1]
se_E5 = a5[:,2]
std_E5 = a5[:,3]
L5 = a5[:,4]
se_L5 = a5[:,5]
std_L5 = a5[:,6]

temp_array5_fitE = []
temp_array5_fitL = []
E5_fit = []
L5_fit = []
for i in range(len(temp_array5)):
     if temp_array5[i] < 10:
          temp_array5_fitE.append(temp_array5[i])
          E5_fit.append(E5[i])
     if temp_array5[i] < 10:
          temp_array5_fitL.append(temp_array5[i])
          L5_fit.append(L5[i])
poptE, pcovE = curve_fit(fit_funcE, temp_array5_fitE, E5_fit)
yvalsE = fit_funcE(temp_array5_fitE, poptE[0],poptE[1])
poptL, pcovL = curve_fit(fit_funcL, temp_array5_fitL, L5_fit)
yvalsL = fit_funcL(temp_array5_fitL, poptL[0],poptL[1])

#temp_array5_SAW = a5_SAW[:,0]
#E5_SAW = a5_SAW[:,1]
#se_E5_SAW = a5_SAW[:,2]
#std_E5_SAW = a5_SAW[:,3]
#L5_SAW = a5_SAW[:,4]
#se_L5_SAW = a5_SAW[:,5]
#std_L5_SAW = a5_SAW[:,6]



#ax1.errorbar(temp_array, E, se_E, fmt='o' , mfc = 'red', label='$J\in[-2,-4]$')
#ax1.errorbar(temp_array1, E1, se_E1, fmt='o' , mfc = 'black', label='$J = -1$')
#ax1.plot(temp_array1_SAW, E1_SAW, 'orange', label='SAW, $J = -1$')
#ax1.errorbar(temp_array3, E3, se_E3, fmt='o' , mfc = 'blue', label='$J = -3$')
ax1.errorbar(temp_array5, E5, se_E5, fmt='o' , mfc = 'black', label='$J = -5$')
ax1.plot(temp_array5_fitE,yvalsE,label='fitting curve, $\\beta=%.2f$'%poptE[1])
#ax1.errorbar(temp_array, E, se_E, fmt='o' , mfc = 'black', label='$J = -7$')

#ax2.errorbar(temp_array, L, se_L, fmt='o' , mfc = 'red', label='$J\in[-2,-4]$')
#ax2.errorbar(temp_array1, L1, se_L1, fmt='o' , mfc = 'black', label='$J = -1$')
#ax2.plot(temp_array1_SAW, L1_SAW, 'orange', label='SAW, $J = -1$')
#ax2.errorbar(temp_array3, L3, se_L3, fmt='o' , mfc = 'blue', label='$J = -3$')
ax2.errorbar(temp_array5, L5, se_L5, fmt='o' , mfc = 'black', label='$J = -5$')
ax2.plot(temp_array5_fitL,yvalsL,label='fitting curve, $\\beta=%.2f$'%poptL[1])
#ax2.errorbar(temp_array, L, se_L, fmt='o' , mfc = 'black', label='$J = -7$')
```

```
    #ax1.set_ylim(-30,0)
    #ax2.set_ylim(1,9)
    ax1.set_xlabel("Temperature", fontsize=20)
    ax1.set_ylabel("Energy", fontsize=20)
    ax1.legend(fontsize=12)
    ax2.set_xlabel("Temperature", fontsize=20)
    ax2.set_ylabel("Length", fontsize=20)
    ax2.legend(fontsize=12)
    plt.suptitle('15 amino acids')
    plt.show()

def plot_30_phase():

    fig = plt.figure()
    ax1 = fig.add_subplot(121)
    ax2 = fig.add_subplot(122)

    a = np.loadtxt('phase_30.txt')
    a1 = np.loadtxt('UniJ1_phase_30.txt')
    a1_SAW = np.loadtxt('SAW_UniJ1_phase_30.txt')
    a3 = np.loadtxt('UniJ3_phase_30.txt')
    a5 = np.loadtxt('UniJ5_phase_30.txt')

    temp_array = a[:,0]
    E = a[:,1]
    se_E = a[:,2]
    std_E = a[:,3]
    L = a[:,4]
    se_L = a[:,5]
    std_L = a[:,6]

    temp_array1 = a1[:,0]
    E1 = a1[:,1]
    se_E1 = a1[:,2]
    std_E1 = a1[:,3]
    L1 = a1[:,4]
    se_L1 = a1[:,5]
    std_L1 = a1[:,6]

    temp_array1_SAW = a1_SAW[:,0]
    E1_SAW = a1_SAW[:,1]
    se_E1_SAW = a1_SAW[:,2]
    std_E1_SAW = a1_SAW[:,3]
    L1_SAW = a1_SAW[:,4]
    se_L1_SAW = a1_SAW[:,5]
    std_L1_SAW = a1_SAW[:,6]

    temp_array3 = a3[:,0]
    E3 = a3[:,1]
    se_E3 = a3[:,2]
    std_E3 = a3[:,3]
    L3 = a3[:,4]
    se_L3 = a3[:,5]
```

```
    std_L3 = a3[:,6]

    temp_array5 = a5[:,0]
    E5 = a5[:,1]
    se_E5 = a5[:,2]
    std_E5 = a5[:,3]
    L5 = a5[:,4]
    se_L5 = a5[:,5]
    std_L5 = a5[:,6]


    #ax1.errorbar(temp_array, E, se_E, fmt='o' , mfc = 'red',
    label='30 amino acids, $J\in[-2,-4]$')
    ax1.errorbar(temp_array1, E1, se_E1, fmt='o' , mfc = 'black'
        , label='$J = -1$')
    ax1.plot(temp_array1_SAW, E1_SAW, 'orange', label='SAW, $J = -1$')
    #ax1.errorbar(temp_array3, E3, se_E3, fmt='o' , mfc = 'blue',
     label='30 amino acids, $J = -3$')
    #ax1.errorbar(temp_array5, E5, se_E5, fmt='o' , mfc = 'green',
     label='30 amino acids, $J = -5$')

    #ax2.errorbar(temp_array, L, se_L, fmt='o' , mfc = 'red',
     label='30 amino acids, $J\in[-2,-4]$')
    ax2.errorbar(temp_array1, L1, se_L1, fmt='o' , mfc = 'black',
     label='$J = -1$')
    ax2.plot(temp_array1_SAW, L1_SAW, 'orange', label='SAW, $J = -1$')
    #ax2.errorbar(temp_array3, L3, se_L3, fmt='o' , mfc = 'blue',
     label='30 amino acids, $J = -3$')
    #ax2.errorbar(temp_array5, L5, se_L5, fmt='o' , mfc = 'green',
     label='30 amino acids, $J = -5$')

    #ax1.set_ylim(-30,0)
    #ax2.set_ylim(1,9)
    ax1.set_xlabel("Temperature", fontsize=20)
    ax1.set_ylabel("Energy", fontsize=20)
    ax1.legend(fontsize=12)
    ax2.set_xlabel("Temperature", fontsize=20)
    ax2.set_ylabel("Length", fontsize=20)
    ax2.legend(fontsize=12)
    plt.suptitle('30 amino acids')
    plt.show()

def plot_100_phase():

    fig = plt.figure()
    ax1 = fig.add_subplot(121)
    ax2 = fig.add_subplot(122)

    a = np.loadtxt('phase_100.txt')
    a1 = np.loadtxt('UniJ1_phase_100.txt')
    a1_SAW = np.loadtxt('SAW_UniJ1_phase_100.txt')
    a3 = np.loadtxt('UniJ3_phase_100.txt')
    a5 = np.loadtxt('UniJ5_phase_100.txt')
```

```
temp_array = a[:,0]
E = a[:,1]
se_E = a[:,2]
std_E = a[:,3]
L = a[:,4]
se_L = a[:,5]
std_L = a[:,6]

temp_array1 = a1[:,0]
E1 = a1[:,1]
se_E1 = a1[:,2]
std_E1 = a1[:,3]
L1 = a1[:,4]
se_L1 = a1[:,5]
std_L1 = a1[:,6]

temp_array1_SAW = a1_SAW[:,0]
E1_SAW = a1_SAW[:,1]
se_E1_SAW = a1_SAW[:,2]
std_E1_SAW = a1_SAW[:,3]
L1_SAW = a1_SAW[:,4]
se_L1_SAW = a1_SAW[:,5]
std_L1_SAW = a1_SAW[:,6]

temp_array3 = a3[:,0]
E3 = a3[:,1]
se_E3 = a3[:,2]
std_E3 = a3[:,3]
L3 = a3[:,4]
se_L3 = a3[:,5]
std_L3 = a3[:,6]

temp_array5 = a5[:,0]
E5 = a5[:,1]
se_E5 = a5[:,2]
std_E5 = a5[:,3]
L5 = a5[:,4]
se_L5 = a5[:,5]
std_L5 = a5[:,6]


#ax1.errorbar(temp_array, E, se_E, fmt='o' , mfc = 'red',
label='100 amino acids, $J\in[-2,-4]$')
ax1.errorbar(temp_array1, E1, se_E1, fmt='o' , mfc = 'black',
 label='$J = -1$')
ax1.plot(temp_array1_SAW, E1_SAW, 'orange', label='SAW, $J = -1$')
#ax1.errorbar(temp_array3, E3, se_E3, fmt='o' , mfc = 'blue',
 label='100 amino acids, $J = -3$')
#ax1.errorbar(temp_array5, E5, se_E5, fmt='o' , mfc = 'green',
 label='100 amino acids, $J = -5$')

#ax2.errorbar(temp_array, L, se_L, fmt='o' , mfc = 'red',
```

```
        label='100 amino acids, $J\in[-2,-4]$')
    ax2.errorbar(temp_array1, L1, se_L1, fmt='o' , mfc = 'black',
        label='$J = -1$')
    ax2.plot(temp_array1_SAW, L1_SAW, 'orange', label='SAW, $J = -1$')
    #ax2.errorbar(temp_array3, L3, se_L3, fmt='o' , mfc = 'blue',
        label='100 amino acids, $J = -3$')
    #ax2.errorbar(temp_array5, L5, se_L5, fmt='o' , mfc = 'green',
        label='100 amino acids, $J = -5$')

    #ax1.set_ylim(-30,0)
    #ax2.set_ylim(1,9)
    ax1.set_xlabel("Temperature", fontsize=20)
    ax1.set_ylabel("Energy", fontsize=20)
    ax1.legend(fontsize=12)
    ax2.set_xlabel("Temperature", fontsize=20)
    ax2.set_ylabel("Length", fontsize=20)
    ax2.legend(fontsize=12)
    plt.suptitle('100 amino acids')
    plt.show()

def plot_E():

    fig = plt.figure()
    ax1 = fig.add_subplot(131)
    ax2 = fig.add_subplot(132)
    ax3 = fig.add_subplot(133)

    a15 = np.loadtxt('phase_15.txt')
    a15_1 = np.loadtxt('UniJ1_phase_15.txt')
    a15_3 = np.loadtxt('UniJ3_phase_15.txt')
    a15_5 = np.loadtxt('UniJ5_phase_15.txt')
    a30 = np.loadtxt('phase_30.txt')
    a30_1 = np.loadtxt('UniJ1_phase_30.txt')
    a30_3 = np.loadtxt('UniJ3_phase_30.txt')
    a30_5 = np.loadtxt('UniJ5_phase_30.txt')
    a100 = np.loadtxt('phase_100.txt')
    a100_1 = np.loadtxt('UniJ1_phase_100.txt')
    a100_3 = np.loadtxt('UniJ3_phase_100.txt')
    a100_5 = np.loadtxt('UniJ5_phase_100.txt')

    temp_array15 = a15[:,0]
    E15 = a15[:,1]
    se_E15 = a15[:,2]

    temp_array15_1 = a15_1[:,0]
    E15_1 = a15_1[:,1]
    se_E15_1 = a15_1[:,2]

    temp_array15_3 = a15_3[:,0]
    E15_3 = a15_3[:,1]
    se_E15_3 = a15_3[:,2]

    temp_array15_5 = a15_5[:,0]
```

```
E15_5 = a15_5[:,1]
se_E15_5 = a15_5[:,2]

ax1.errorbar(temp_array15, E15, se_E15, fmt='o' , mfc = 'red',
 label='$J\in[-2,-4]$')
ax1.errorbar(temp_array15_1, E15_1, se_E15_1, fmt='o' ,
 mfc = 'black', label='$J = -1$')
ax1.errorbar(temp_array15_3, E15_3, se_E15_3, fmt='o' ,
 mfc = 'blue', label='$J = -3$')
ax1.errorbar(temp_array15_5, E15_5, se_E15_5, fmt='o' ,
 mfc = 'green', label='$J = -5$')


temp_array30 = a30[:,0]
E30 = a30[:,1]
se_E30 = a30[:,2]

temp_array30_1 = a30_1[:,0]
E30_1 = a30_1[:,1]
se_E30_1 = a30_1[:,2]

temp_array30_3 = a30_3[:,0]
E30_3 = a30_3[:,1]
se_E30_3 = a30_3[:,2]

temp_array30_5 = a30_5[:,0]
E30_5 = a30_5[:,1]
se_E30_5 = a30_5[:,2]

ax2.errorbar(temp_array30, E30, se_E30, fmt='o' ,
 mfc = 'red', label='$J\in[-2,-4]$')
ax2.errorbar(temp_array30_1, E30_1, se_E30_1, fmt='o' ,
 mfc = 'black', label='$J = -1$')
ax2.errorbar(temp_array30_3, E30_3, se_E30_3, fmt='o' ,
 mfc = 'blue', label='$J = -3$')
ax2.errorbar(temp_array30_5, E30_5, se_E30_5, fmt='o' ,
 mfc = 'green', label='$J = -5$')

temp_array100 = a100[:,0]
E100 = a100[:,1]
se_E100 = a100[:,2]

temp_array100_1 = a100_1[:,0]
E100_1 = a100_1[:,1]
se_E100_1 = a100_1[:,2]

temp_array100_3 = a100_3[:,0]
E100_3 = a100_3[:,1]
se_E100_3 = a100_3[:,2]

temp_array100_5 = a100_5[:,0]
E100_5 = a100_5[:,1]
se_E100_5 = a100_5[:,2]
```

```
475
        ax3.errorbar(temp_array100, E100, se_E100, fmt='o' ,
         mfc = 'red', label='$J\in[-2,-4]$')
        ax3.errorbar(temp_array100_1, E100_1, se_E100_1, fmt='o' ,
         mfc = 'black', label='$J = -1$')
480     ax3.errorbar(temp_array100_3, E100_3, se_E100_3, fmt='o' ,
         mfc = 'blue', label='$J = -3$')
        ax3.errorbar(temp_array100_5, E100_5, se_E100_5, fmt='o' ,
         mfc = 'green', label='$J = -5$')

485     ax1.set_xlabel("Temperature", fontsize=20)
        ax1.set_ylabel("Energy", fontsize=20)
        ax1.set_title("15 amino acids")
        ax1.legend(fontsize=12)
        ax2.set_xlabel("Temperature", fontsize=20)
490     #ax2.set_ylabel("Energy", fontsize=20)
        ax2.legend(fontsize=12)
        ax2.set_title("30 amino acids")
        ax3.set_xlabel("Temperature", fontsize=20)
        #ax3.set_ylabel("Energy", fontsize=20)
495     ax3.legend(fontsize=12)
        ax3.set_title("100 amino acids")
        plt.show()

    def plot_SAW_compare():
500     fig = plt.figure()
        ax1 = fig.add_subplot(121)
        ax2 = fig.add_subplot(122)

        a15_1 = np.loadtxt('UniJ1_phase_15.txt')
505     a15_1_SAW = np.loadtxt('SAW_UniJ1_phase_15.txt')
        a30_1 = np.loadtxt('UniJ1_phase_30.txt')
        a30_1_SAW = np.loadtxt('SAW_UniJ1_phase_30.txt')
        a100_1 = np.loadtxt('UniJ1_phase_100.txt')
        a100_1_SAW = np.loadtxt('SAW_UniJ1_phase_100.txt')
510
        temp_array15_1 = a15_1[:,0]
        E15_1 = a15_1[:,1]
        se_E15_1 = a15_1[:,2]
        L15_1 = a15_1[:,4]
515     se_L15_1 = a15_1[:,5]
        temp_array15_1_SAW = a15_1_SAW[:,0]
        E15_1_SAW = a15_1_SAW[:,1]
        se_E15_1_SAW = a15_1_SAW[:,2]
        L15_1_SAW = a15_1_SAW[:,4]
520     se_L15_1_SAW = a15_1_SAW[:,5]

        temp_array30_1 = a30_1[:,0]
        E30_1 = a30_1[:,1]
        se_E30_1 = a30_1[:,2]
525     L30_1 = a30_1[:,4]
        se_L30_1 = a30_1[:,5]
        temp_array30_1_SAW = a30_1_SAW[:,0]
```

```
    E30_1_SAW = a30_1_SAW[:,1]
    se_E30_1_SAW = a30_1_SAW[:,2]
    L30_1_SAW = a30_1_SAW[:,4]
    se_L30_1_SAW = a30_1_SAW[:,5]

    temp_array100_1 = a100_1[:,0]
    E100_1 = a100_1[:,1]
    se_E100_1 = a100_1[:,2]
    L100_1 = a100_1[:,4]
    se_L100_1 = a100_1[:,5]
    temp_array100_1_SAW = a100_1_SAW[:,0]
    E100_1_SAW = a100_1_SAW[:,1]
    se_E100_1_SAW = a100_1_SAW[:,2]
    L100_1_SAW = a100_1_SAW[:,4]
    se_L100_1_SAW = a100_1_SAW[:,5]

    ax1.errorbar(temp_array15_1, E15_1, se_E15_1, fmt='o' ,
     mfc = 'black', label='15 amino acids')
    ax1.plot(temp_array15_1_SAW, E15_1_SAW, 'orange',
     label='15 amino acids, SAW')
    ax1.errorbar(temp_array30_1, E30_1, se_E30_1, fmt='o' ,
     mfc = 'blue', label='30 amino acids')
    ax1.plot(temp_array30_1_SAW, E30_1_SAW, 'darkblue',
     label='30 amino acids, SAW')
    ax1.errorbar(temp_array100_1, E100_1, se_E100_1, fmt='o' ,
     mfc = 'green', label='100 amino acids')
    ax1.plot(temp_array100_1_SAW, E100_1_SAW, 'darkgreen',
     label='100 amino acids, SAW')

    ax2.errorbar(temp_array15_1, L15_1, se_L15_1, fmt='o' ,
     mfc = 'black', label='15 amino acids')
    ax2.plot(temp_array15_1_SAW, L15_1_SAW, 'orange',
     label='15 amino acids, SAW')
    ax2.errorbar(temp_array30_1, L30_1, se_L30_1, fmt='o' ,
     mfc = 'blue', label='30 amino acids')
    ax2.plot(temp_array30_1_SAW, L30_1_SAW, 'darkblue',
     label='30 amino acids, SAW')
    ax2.errorbar(temp_array100_1, L100_1, se_L100_1, fmt='o' ,
     mfc = 'green', label='100 amino acids')
    ax2.plot(temp_array100_1_SAW, L100_1_SAW, 'darkgreen',
     label='100 amino acids, SAW')

    ax1.set_xlabel("Temperature", fontsize=20)
    ax1.set_ylabel("Energy", fontsize=20)
    ax1.legend(fontsize=12)
    ax2.set_xlabel("Temperature", fontsize=20)
    ax2.set_ylabel("Length", fontsize=20)
    ax2.legend(fontsize=12)
    plt.suptitle('$J = -1$')
    plt.show()

def main():

```

```
    #plot_15_phase()
    #plot_30_phase()
    #plot_100_phase()
    #plot_E()
    #plot_SAW_compare()
    analysis_phase_transition()
    plot_fit_phasetransition()

if __name__ == '__main__':
    main()
```

E Appendix: Figures of simulation results of the protein with 15, 30, 100 amino acids

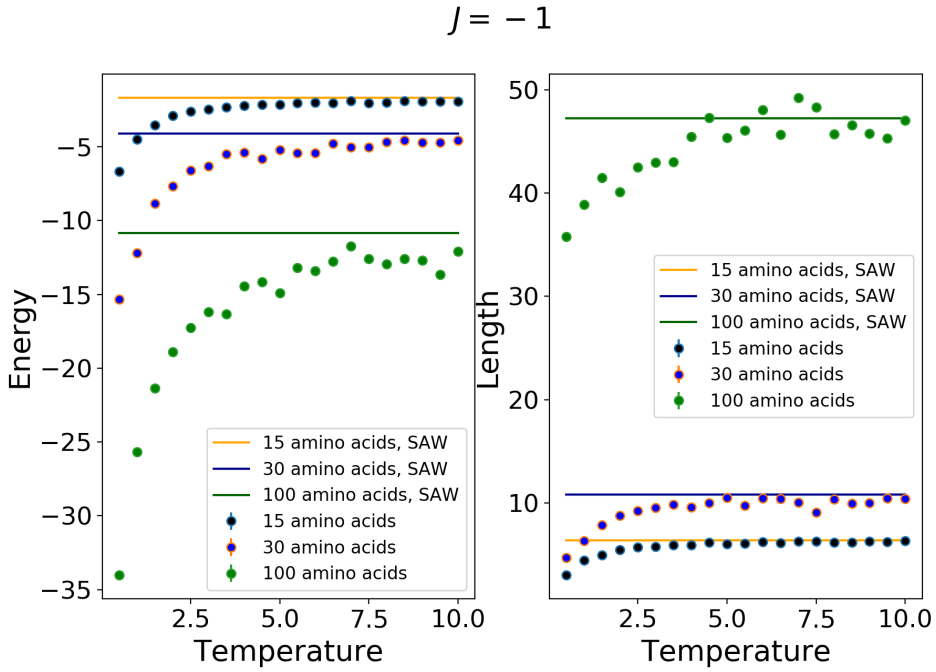

Figure 1: The comparison of self-avoiding random walk simulation and protein folding simulation for chains with 15, 30, 100 amino acids. The interaction coefficient J is -1.

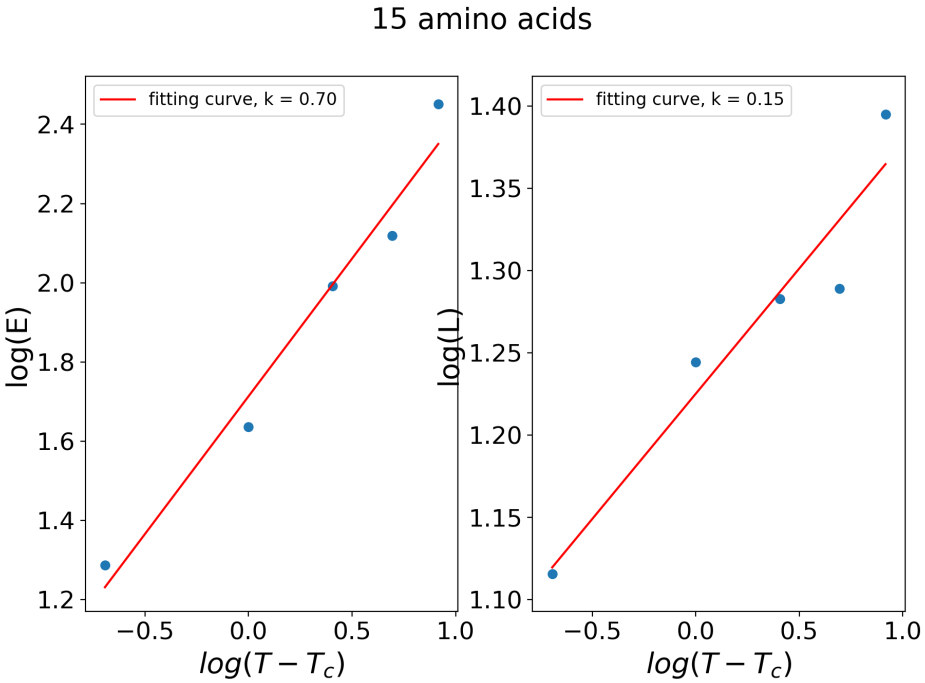

Figure 2: The linear fitting results of $\ln(E)$, $\ln(L)$ vs $\ln(T - T_c)$ for the protein system with 15 amino acids. The interaction coefficient J is -5.

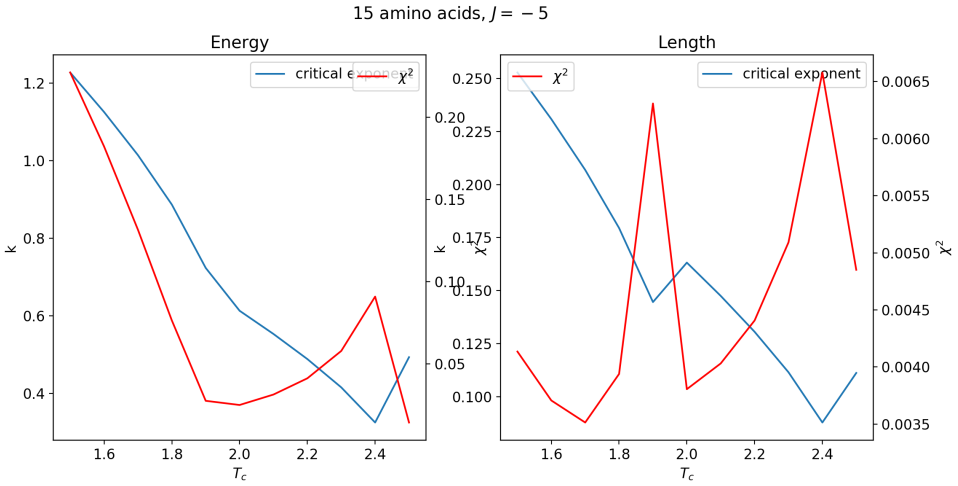

Figure 3: The linear fitting results of $\ln(E)$, $\ln(L)$ vs $\ln(T - T_c)$ for the protein system with 15 amino acids and χ^2 plot for different T_c. The interaction coefficient J is -5.

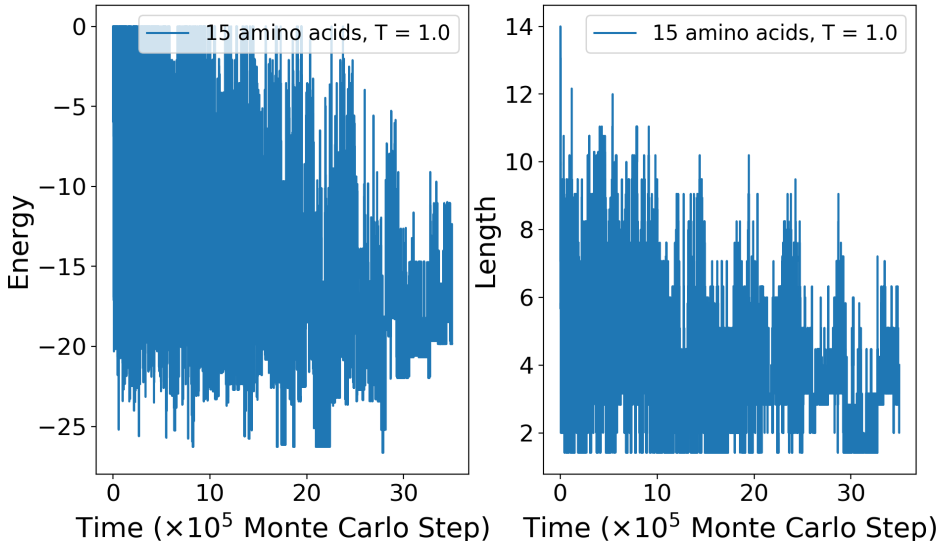

Figure 4: Simulation results for the evolution of the energy and end-to-end distance (length) of the protein consisting of 15 amino acids in low temperature. Simulated annealing method is applied in this simulation.

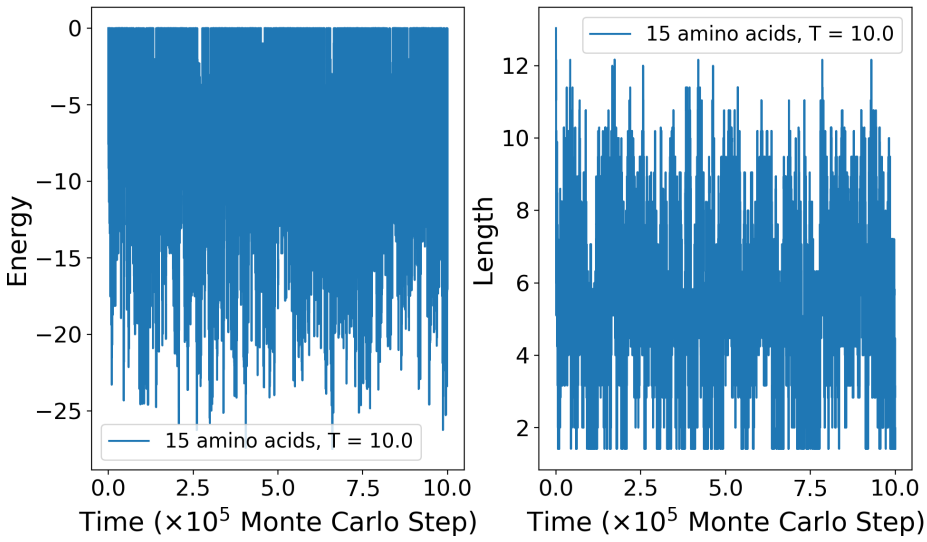

Figure 5: Simulation results for the evolution of the energy and end-to-end distance (length) of the protein consisting of 15 amino acids in high temperature.

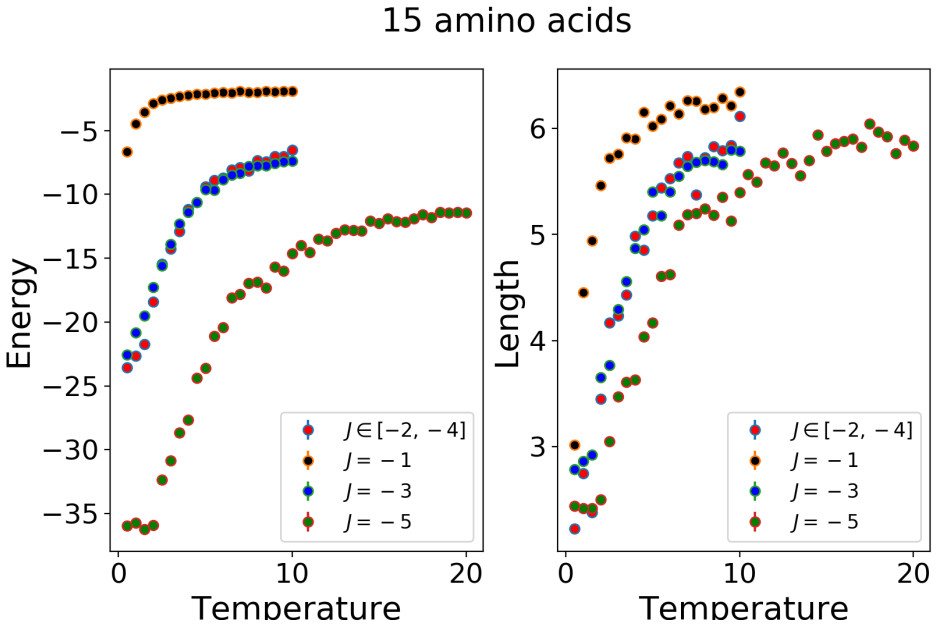

Figure 6: Simulation results for the relationship between temperature and the energy (left) and end-to-end distance (right) of the protein consisting of 15 amino acids with different interaction coefficient J.

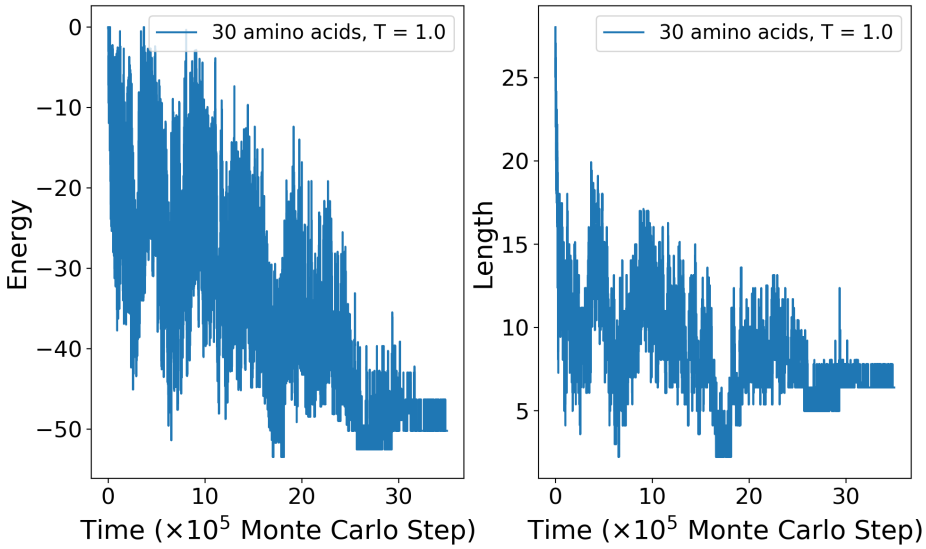

Figure 7: Simulation results for the evolution of the energy and end-to-end distance (length) of the protein consisting of 30 amino acids in low temperature. Simulated annealing method is applied in this simulation.

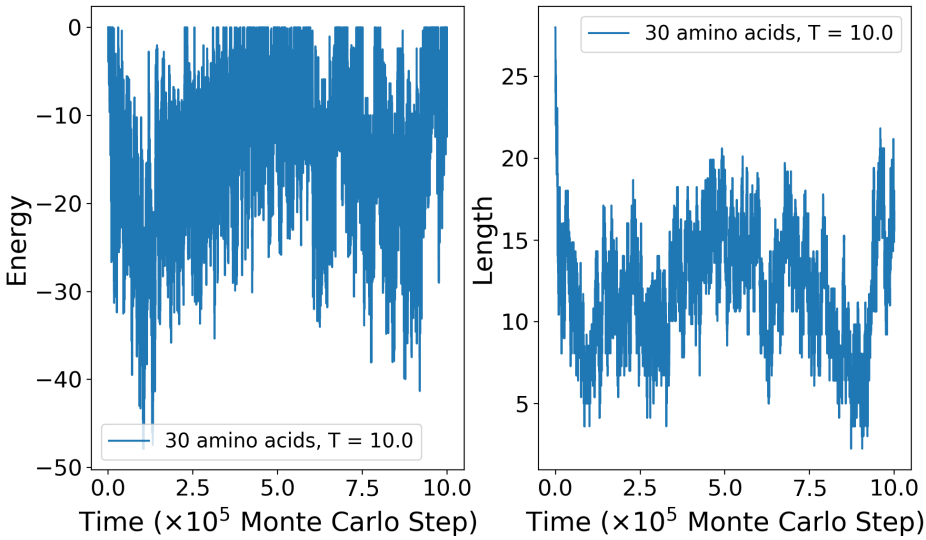

Figure 8: Simulation results for the evolution of the energy and end-to-end distance (length) of the protein consisting of 30 amino acids in high temperature.

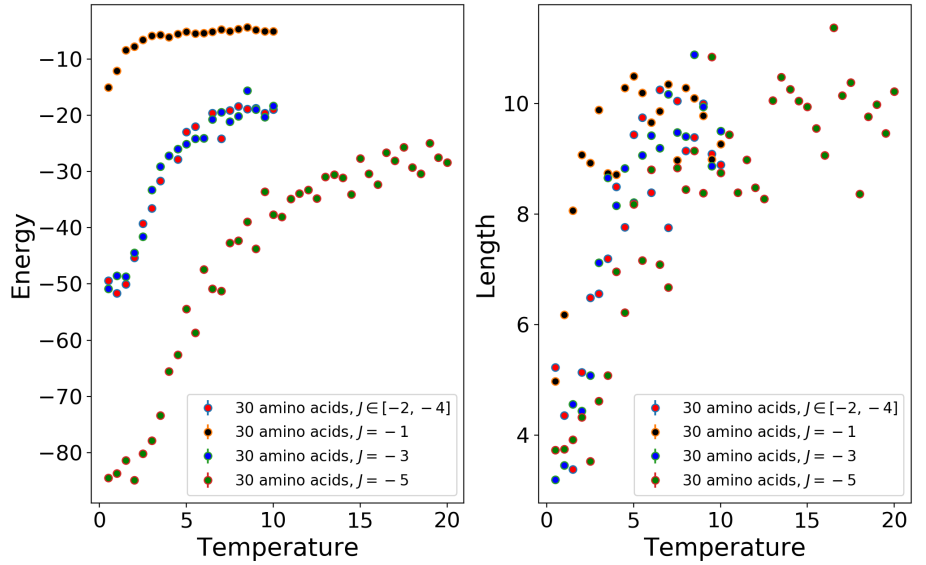

Figure 9: Simulation results for the relationship between temperature and the energy (left) and end-to-end distance (right) of the protein consisting of 30 amino acids with different interaction coefficient J.

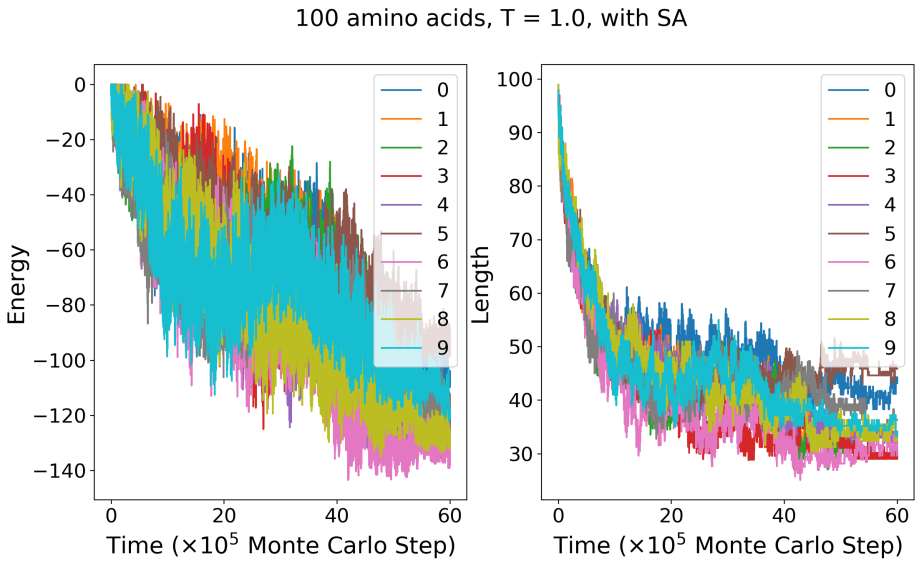

Figure 10: Simulation results for the evolution of the energy and end-to-end distance (length) of the protein consisting of 100 amino acids in low temperature. Simulation has been run by 10 times. Simulated annealing method is applied in this simulation.

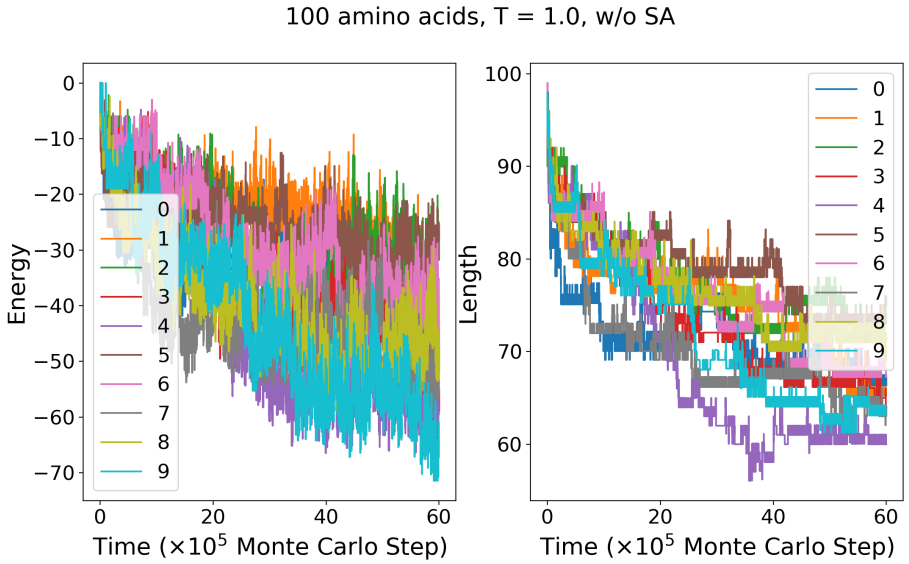

Figure 11: Simulation results for the evolution of the energy and end-to-end distance (length) of the protein consisting of 100 amino acids in low temperature. Simulation has been run by 10 times.

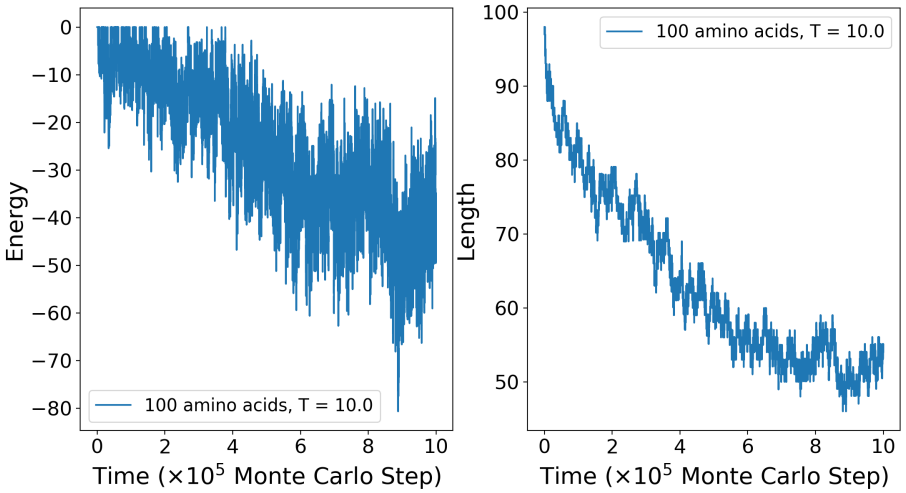

Figure 12: Simulation results for the evolution of the energy and end-to-end distance (length) of the protein consisting of 100 amino acids in high temperature.

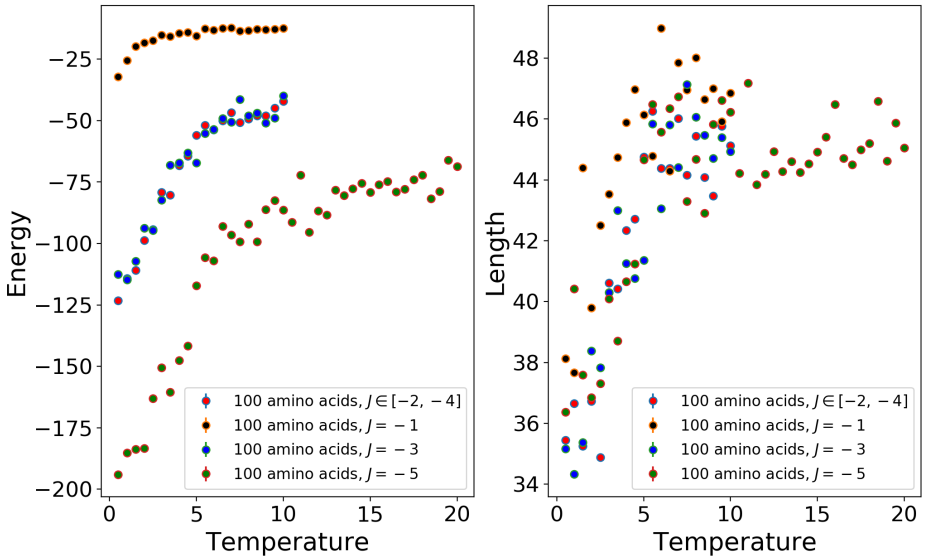

Figure 13: Simulation results for the relationship between temperature and the energy (left) and end-to-end distance (right) of the protein consisting of 100 amino acids with different interaction coefficient J..