Stat 525 Final Project: Sequential Importance Sampling with Pilot-exploration Resampling

Zhikun Cai,[1] Chenchao Shou,[2] and Guanfeng Gao[1]

[1)] *Department of Nuclear, Plasma, and Radiological Engineering*

[2)] *Department of Mechanical Sciences and Engineering*

(Dated: 11 May 2016)

The prediction of 3D protein structure from the 1D amino acid sequence is a challenging problem, even in the simplified 2D hydrophobic-hydrophilic (HP) lattice model. To address this challenge, multiple advanced Monte Carlo algorithms have been proposed in literature. Among these, a sequential importance sampling with pilot-exploration resampling (SISPER) algorithm[1] has been proved efficient in improving the sampling of the lowest-energy conformations of proteins in a 2D HP model. In this project, we implemented an in-house code of the SISPER algorithm and applied it to search for the native states of five protein HP sequences in a two-dimensional lattice. Compared to the traditional sequential importance sampling with multi-step lookahead, SISPER collects the future information to guide the conformation growth so that the probability of constructing valid conformations is significantly improved. We also explored the temperature effect by performing the searching at various temperatures. As a result, we have successfully found the lowest-energy conformations for three protein sequences. For the other two sequences, only the conformations with the second lowest energy were obtained. The discrepancies between our results and the originally published results may lie in the resampling steps, where we used the simple random sampling method instead of residual resampling method.

I. INTRODUCTION

The protein folding problem has been an intensive research topic for almost half a century. Scientists have been aiming to answer three key quesitons[2]: (1) What is the physical rule that determines the 3D native folded state of a protein from its 1D amino acid sequence? (2) How can a polypeptide chain fold so fast into its native state in spite of a trenmendous set of possible conformations? (3) Is it possible to devise some computer algorithms to predict the native folded structure from its amino acid squence?

The prediction of protein structure is a computational challenge even to computers. The number of possible conformations of a protein grows exponentially with the length of amino acid sequence. Searching for the native state, the lowest-energy conformation is often frustrated by numerous local energy minima on the rugged energy landscape of a protein. To gain some insights into this problem, researchers have turned to the simplified 2D protein hydrophobic-hydrophilic (HP) lattice model. Besides, to accelerate the search for the lowest-energy conformation of a protein, multiple new Monte Carlo algorithms have been proposed, including the genetic algorithm[3], evolutionary Monte Carlo[4], pruned-enriched Rosenbluth method (PERM)[5], and sequential importance sampling with pilot-exploration resampling (SISPER)[1].

In this project, we applied the SISPER algorithm proposed by Zhang et al.[1] to search for the native states of proteins in a 2D HP model. Compared to the traditional sequential importance sampling with multi-step lookahead, SISPER uses the pilot-exploration as a means to collect future information about conformations so that the probability of constructing valid ones is significantly improved. This report is organized as follows. Section II introduces the 2D HP model. Section III describes the SISPER algorithm, which is followed by the numerical results and conclusion in Section IV and V, respectively.

II. THE 2D HP MODEL

In the 2D HP model, a protein is considered as a chain of hydrophobic (H) and hydrophilic (P) residues. A conformation of protein is defined as a chain of adjacent sites, growing sequentially by a self-avoiding walk on a two-dimensional square lattice. Each lattice site

is occupied only once for one conformation. Residues interact only with their nonbonded nearest neighbors via a simple energy function: $\epsilon_{\mathrm{HH}} = -1$ and $\epsilon_{\mathrm{PP}} = \epsilon_{\mathrm{HP}} = 0$.

In this report, a conformation of a protein sequence of length d is represented by a list of 2D points $\mathbf{x}_d = (x_1, x_2, \ldots, x_d)$, where each x_i denotes a 2D point. The energy of a conformation, $U_d(\mathbf{x}_d)$, is the sum of all pairwise energies. At a certain temperature τ, the probability for a protein sequence to occupy a certain conformation follows the Boltzmann distribution,

$$\pi_d(\mathbf{x}_d) \propto \exp(-U_d(\mathbf{x}_d)/\tau). \tag{1}$$

We denote the marginal distribution of $\mathbf{x}_d$ based on the distribution $\pi_{d+k}(\mathbf{x}_{d+k})$ as $\pi_{d+k}(\mathbf{x}_d)$.

III. ALGORITHMS

A. δ-step Lookahead

In the self-avoiding walk process, samples can be drawn effectively by looking ahead δ-steps so that the probability a conformation terminates prematurely before reaching its desired length is greatly reduced. Given a current conformation $\mathbf{x}_{t-1}$, the δ-step lookahead proposes a new move based on the conditional distribution function

$$q_t(x_t|\mathbf{x}_{t-1}) = \pi_{t+\delta-1}(x_t|\mathbf{x}_{t-1}) \equiv \frac{\sum_{x_{t+1},\ldots,x_{t+\delta-1}} \exp\{-U_{t+\delta-1}(\mathbf{x}_{t-1}, x_t, \ldots, x_{t+\delta-1})/\tau\}}{\sum_{x_t'} \sum_{x_{t+1},\ldots x_{t+\delta-1}} \exp\{-U_{t+\delta-1}(\mathbf{x}_{t-1}, x_t', \ldots, x_{t+\delta-1})/\tau\}} \tag{2}$$

The larger the δ value is, the more future information is used to inform the next step. Therefore, it's more likely to generate a valid sample. However, larger δ value means more computational efforts so there is a trade-off between computational cost and performance in selecting δ value.

B. Sequential Importance Sampling (SIS)

Sequential importance sampling (SIS) is a specific type of importance sampling methods, in which a high-dimensional sample is drawn sequentially from some prescribed one-dimensional distributions. The samples resulted from SIS typically do not follow the target distribution.

To correct this bias, an appropriate weight has to be assigned to each generated sample. Specifically, SIS generates samples and their weights through the following steps:

- Step 1: Set $w_0 = 1$ at $t = 1$ for all the samples x_1^j $(j = 1, ..., N)$.
- Step 2: Draw x_t from some conditional distribution $g_t(\cdot|x_1, ..., x_{t-1})$ for each j.
- Step 3: Compute the incremental weight $u_t = \frac{\pi_t(x_1,...,x_t)}{\pi_{t-1}(x_1,...,x_{t-1})g_t(x_t|x_1,...x_{t-1})}$ and update the weight $w_t = w_{t-1}u_t$ for each j, where π_t is the target distribution. Then go to Step 2.

Given the generated weights, the mean estimator of $h(x_t)$ takes the form

$$\widehat{h}(x_t) = \frac{\sum_{j=1}^N w_j h(x_t^j)}{\sum_{j=1}^N w_j}. \tag{3}$$

C. Pilot-Exploration Resampling (PER)

Pilot-Exploration Resampling (PER) is a resampling strategy proposed by Zhang et al.[1] to improve the sampling of the lowest-energy conformation. It determines the resampling probabilities by sending out a pilot exploration "team" to spy on the future information. The PER algorithm is given as follows:

- Step 1: Given a set of current conformation $S_t = \{\mathbf{x}_t^j, j = 1, ..., N\}$, build the next Δ residues m independent times for each conformation $\mathbf{x}_t^j$ in S_t by a ρ-step lookahead method; this generates $\{x_{t+1}^{j,l}, \ldots, x_{t+\Delta}^{j,l}, l = 1, \ldots, m\}$
- Step 2: For each pilot path l of conformation $\mathbf{x}_t^j$, compute the Boltzmann weight with ρ-step lookahead using the marginal distribution $b_t^{j,l} = \pi_{t+\Delta+\rho-1}(\mathbf{x}_t^{j,l}, x_{t+1}^{j,l}, ..., x_{t+\Delta}^{j,l})$
- Step 3: For each conformation $\mathbf{x}_t^j$, compute the resampling probability $a_t^j = (\frac{1}{m}\sum_{l=1}^m b_t^{j,l})^\alpha$, $j = 1, \ldots, N$
- Step 4: Resample a new conformation set from S_t with the probability vector proportional to $\{a_t^1, \ldots, a_t^N\}$

D. SISPER

SISPER uses the SIS for growing conformations and the PER for resampling conformations. In a growing step, the SIS with δ-step lookahead is used to place the next residue; after every several growing steps, PER is employed to decide the resampling probabilities and then resample a new conformation set. The algorithm is shown as follows:

- Step 1: Prepare a set of initial conformations $S_t = \{\mathbf{x}_t^j,\ t = 2,\ j = 1, \ldots, N\}$, where each conformation $\mathbf{x}_t^j$ is initialized with the first two points $(0, 0)$ and $(0, 1)$; Initialize all the weights w_t^j by 1.

- Step 2: For each partial conformation $\mathbf{x}_t^j$, use the δ-step look-ahead method to decide the next point x_{t+1}^j, update the new conformation $\mathbf{x}_{t+1}^j = (\mathbf{x}_t^j, x_{t+1}^j)$ and the weight w^j.

  (1) compute the conditional distrubitoion $q_t^j(x_{t+1}|\mathbf{x}_t^j)$ using δ-step lookahead

  (2) generate x_{t+1}^j from $q_t^j(x_{t+1}|\mathbf{x}_t^j)$ and update the weight $w_{t+1}^j = w_t^j u_t^j$, where the incremental weight $u_t^j = \frac{\pi_{t+1}(\mathbf{x}_{t+1}^j)}{\pi_{t+1}(\mathbf{x}_{t+1}^j) q_t^j(x_{t+1}|\mathbf{x}_t^j)}$

- Step 3: Let $t = t + 1$,

  (1) if t reaches the length of the whole sequence, go to Step4

  (2) if t is the multiple of λ, perform a resampling step from S_t using the PER strategy

  (3) otherwise, go to Step2

- Step 4: Sampling terminates; output the final conformations, weights, and energies

IV. NUMERICAL RESULTS

We test our SISPER algorithm on five 2D HP sequences listed in Table I, with $N = 5000$, $\delta = 5$, $\lambda = 2$, $m = 20$, $\Delta = 20$, $\rho = 1$ and $\alpha = 0.5$. These parameters along with HP sequences are all taken from Ref. 1. We vary the parameter τ from 0.1 to 1 with increment of 0.1[6]. The trade-off of selecting τ is that for small τ, the probability of the conformation with ground energy state is significantly higher than that of other conformation but the

TABLE I. 2D HP test sequences

Sequence No.	Length[a]	Sequence[b]
1	20	HPHPPHHPHPPHPHHPPHPH
2	36	PPPHHPPHHPPPPPHHHHHHHPPHHPPPPHHPPHPP
3	48	PPHPPHHPPHHPPPPPHHHHHHHHHHPPPPPPHHPPHHPP HPPHHHHH
4	50	HHPHPHPHPHHHHPHPPPHPPPHPPPPHPPPHPPPHPHHH HPHPHPHPHH
5	60	PPHHHPHHHHHHHHPPPHHHHHHHHHHPHPPPHHHHHHHH HHHHPPPPHHHHHHHPHHPHP

[a] The length denotes the number of residues of the sequence

[b] All the sequences are taken from Ref. 4

probability landscape becomes more spiky as τ decreases so it's more likely to get stuck in the local minima for particularly small τ. We test SISPER for different τ in the hope of finding appropriate value of τ for each sequence. The code implementing our SISPER algorithm is appended to this report (see Appendix 1).

Table II lists the lowest energy for each sequence found by our SISPER algorithm, compared to the theoretical ground energy as well as the lowest energy found by the original SISPER algorithm. As we can see, our algorithm successfully finds the ground energy for sequence 1, 2 and 5 but fails on sequence 3 and 4. The discrepancy between the result of our SISPER algorithm and that of the original SISPER algorithm could be due to different ways of resampling. Indeed, Zhang et. al. [1] suggests two resampling methods, namely simple random resampling and residue resampling. For our SISPER algorithm, we choose the simple random resampling method, which might not be the actual one chosen by Zhang et. al. Figure 1, 2, 3, 4 and 5 show the conformations of the lowest energies for sequence 1, 2, 3, 4 and 5 respectively. For the complete set of conformations for each τ and each HP sequence, please refer to the figures in the attachment.

TABLE II. Performance of Our SISPER Algorithm

Sequence No.	Ground energy[a]	Our SISPER [b]	SISPER[c]
1	-9	-9	-9
2	-14	-14	-14
3	-23	-22	-23
4	-21	-20	-21
5	-36	-36	-36

[a] The putative ground energies are recorded in Ref. 4

[b] The τ values of sequence 1, 2, 4 and 5 are 0.5. The τ value of sequence 3 is 0.4.

[c] original SISPER algorithm implemented in Ref. 1

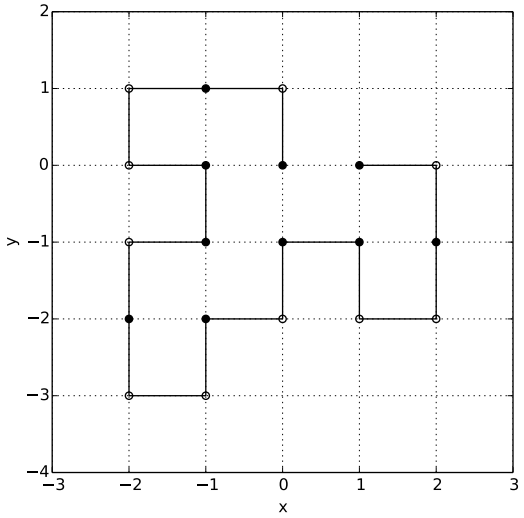

FIG. 1. A conformation of putative ground energy -9 for sequence 1 found by our SISPER algorithm with $\tau = 0.5$

V. CONCLUSION

In this project, we implement sequential importance sampling with pilot-exploration resampling (SISPER) algorithm proposed by Zhang et. al[1] to search for the ground energy state of protein that lives on a two-dimensional hydrophobic-hydrophlic (HP) lattice. Compared to the traditional multi-step look ahead sequential importance sampling, SISPER uses

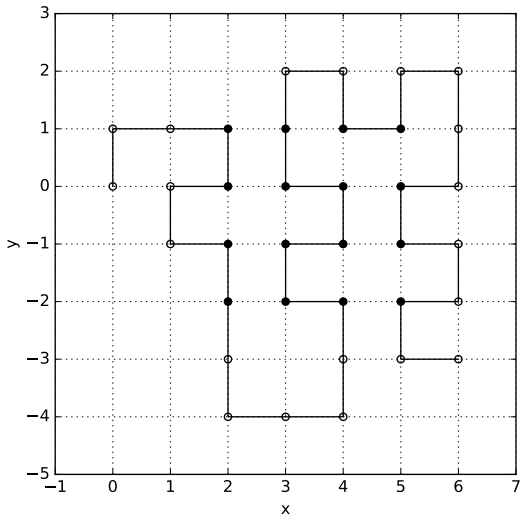

FIG. 2. A conformation of putative ground energy -14 for sequence 2 found by our SISPER algorithm with $\tau = 0.5$

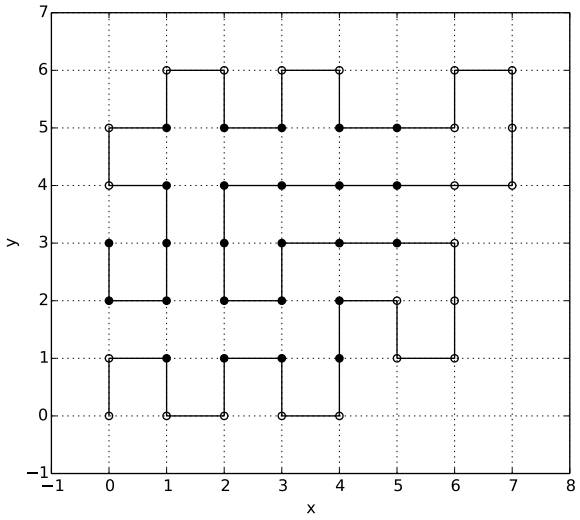

FIG. 3. A conformation of energy -22 for sequence 3 found by our SISPER algorithm with $\tau = 0.4$

pilot-exploration as a means to collect future information about conformations so that the probability of constructing valid ones is significantly improved. Our SISPER algorithm is tested on the five HP sequences given in Ref. 1 and successfully finds the ground energy for sequence 1, 2 and 5 but unfortunately fails on sequence 3 and 4. The discrepancy between

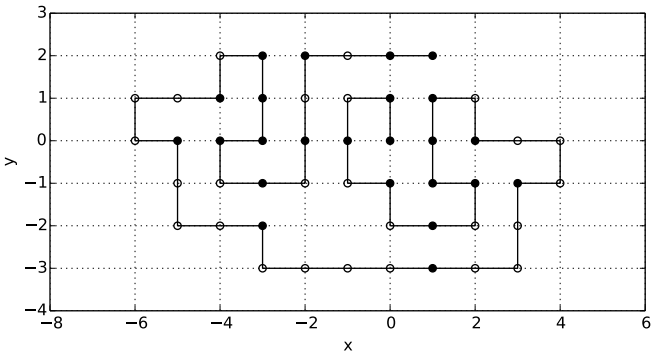

FIG. 4. A conformation of energy -20 for sequence 4 found by our SISPER algorithm with $\tau = 0.5$

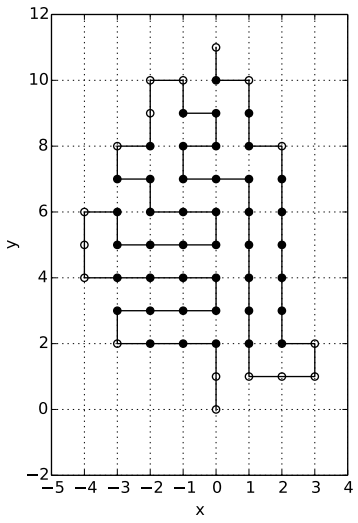

FIG. 5. A conformation of putative ground energy -36 for sequence 5 found by our SISPER algorithm with $\tau = 0.5$

the result of our SISPER algorithm and that of the original SISPER algorithm could be due to different ways of resampling. Exploring the impact of different resampling methods on the algorithm performance might be interesting and could be one direction of future work.

REFERENCES

[1]J. L. Zhang and J. S. Liu, The Journal of Chemical Physics **117**, 3492 (2002).

[2]K. A. Dill and J. L. MacCallum, Science (New York, N.Y.) **338**, 1042 (2012).

[3]R. Unger and J. Moult, Journal of Molecular Biology **231**, 75 (1993).

[4]F. Liang and W. H. Wong, The Journal of Chemical Physics **115**, 3374 (2001).

[5]U. Bastolla, H. Frauenkron, E. Gerstner, P. Grassberger, and W. Nadler, PROTEINS: Structure, Function, and Genetics **32**, 52 (1998).

[6]Given the available computing power, we are only able to test $\tau = 0.5$ for sequence 5.

Appendix: Code

1. sisper.py

```
#
# STAT525-SISPER, 2016
#
# Problem:
# Searching for the lowest-energy folded state of protein in a 2D hydrophobic-hydrophilic
    (HP) lattice model
#
# Algorithm:
# Sequential Importance Sampling with Pilot-Exploration Resampling (SISPER)
#
# Reference:
# J. L. Zhang and J. S. Liu, A new sequential improtance sampling method and its
   application to the two-dimensional hydrophobic-hydrophilic model,
# Journal of Chemical Physics, 117, 3492 (2002)
#
# Developers:
#  Zhikun Cai, Chenchao Shou, Guanfeng Gao
#

# Usage:
# $ python sisper.py sequence_file_name output_file_name tau


# import modules
from __future__ import division
from scipy import stats
```

```
import numpy as np
import random
import sys
import time



# initialize global parameters
N = 5000 # num of conformations
delta = 5 # num of steps lookahead in the regular SIS steps
lamb = 2 # frequency of resampling
m = 20 # num of independent paths in the resampling steps
Delta = 20 # num of steps explored in the resampling steps
rho = 1 # num of steps lookahead in the resampling steps
alpha = 0.5 # power of the Boltzmann weight in calculating the resampling probability
#tau = 0.5 # temperature

# define global constants
res_H = 1 # flag indicating the residue type, H or P
res_P = 0

esp_HH = -1 # pairwised energy
esp_HP = 0
esp_PP = 0

left = 0 # flag indicating the torsion angle
right = 1
ahead = 2
```

```
# define functions
def read_input_sequence(sequence_file_name):
    """
    read in the input HP sequence from file
    """
    input_HP_sequence = ''
    with open(sequence_file_name, 'r') as file_id:
        for line in file_id:
            input_HP_sequence += line.replace('\n', '')
    return input_HP_sequence



def write_conformations(S, w, U, t_exec, output_file_name):
    """
    write the conformations to file from the lowest energy to the highest
    and the execution time
    """
    sorted_energy_indice = np.argsort(U)
    U_sorted = []
    S_sorted = []
    w_sorted = []
    for index in sorted_energy_indice:
        U_sorted.append(U[index])
        S_sorted.append(S[index])
        w_sorted.append(w[index])

    np.savez(output_file_name, energies=U_sorted, coordinates=S_sorted, \
        weights=w_sorted, t_exec=t_exec)
```

```
def compute_couples(input_HP_sequence):
    """
    compute all the possible couples in the series
    """
    num=0 #initialize the size of couples
    couples=np.zeros([len(input_HP_sequence)**2,2])
    for ii in range(len(input_HP_sequence)):
        if input_HP_sequence[ii]=='H':
            for jj in range(ii+3,len(input_HP_sequence),2): #loop through the rest
                possible nodes
                if input_HP_sequence[jj]=='H':
                    couples[num,0]=ii
                    couples[num,1]=jj
                    num=num+1
    couples=couples[0:num,:] #get the effective couples
    return couples



def compute_energy(x, couples):
    """
    Compute the energy of a specific conformation
    Input: current residue position x = [[x1,y1],[x2,y2]...[xn,yn]]
           couples = 2d numpy array
    """
    U=0
    for ii in range(couples.shape[0]):

        if couples[ii,0]>len(x)-1 or couples[ii,1]>len(x)-1:#exclude the couples exceed
            the current set
            continue
```

```
        else:
            point1=np.array(x[int(couples[ii,0])])
            point2=np.array(x[int(couples[ii,1])])
            if np.dot(point1-point2,point1-point2)==1: #neighbour point
                U=U+esp_HH
    return U


def one_step(x):
    """
    return all possible configurations after one step given the current position
    and direction for each configuration

    Input: current residue position x = [[x1,y1],[x2,y2]...[xn,yn]]
    Output: [a list of configurations, a list of directions]
    """
    assert (len(x)>=2)
    nbrs,dirs = neighbor(x[-1],x[-2]) # find 3 neighbors of the end point and their
        directions
    x3 = x[:-3] # excluding the last 3 points
    configs = []
    dir_configs = []
    for i,nb in enumerate(nbrs):
        d = dirs[i]
        valid_nb = True
        for pt in x3:
            if pt == nb:
                valid_nb = False
        if valid_nb:
            configs.append(x+[nb])
```

```
            dir_configs.append(d)
    return [configs,dir_configs]



def neighbor(pt,cpt):
    """
    return the neighbors of pt other than cpt as well as direction

    Input: pt = [x1,y1], cpt = [x2,y2]
    Output: [a list of 3 points, a list of 3 directions]

    """
    nbrs = []
    dirs = []
    if pt[0] == cpt[0]:
        if pt[1] == cpt[1]+1:
            nbrs = [[pt[0]+1,pt[1]],[pt[0]-1,pt[1]],[pt[0],pt[1]+1]]
            dirs = [right,left,ahead]
        elif pt[1] == cpt[1]-1:
            nbrs = [[pt[0]+1,pt[1]],[pt[0]-1,pt[1]],[pt[0],pt[1]-1]]
            dirs = [left,right,ahead]
    if pt[1] == cpt[1]:
        if pt[0] == cpt[0]+1:
            nbrs = [[pt[0]+1,pt[1]],[pt[0],pt[1]+1],[pt[0],pt[1]-1]]
            dirs = [ahead,left,right]
        elif pt[0] == cpt[0]-1:
            nbrs = [[pt[0]-1,pt[1]],[pt[0],pt[1]+1],[pt[0],pt[1]-1]]
            dirs = [ahead,right,left]
    assert (len(nbrs) == 3 and len(dirs) == 3), 'error! number of neighbors is invalid!'
        # sanity check
```

```
    return [nbrs,dirs]


def multi_step_look_ahead(x, input_HP_sequence, steps_tmp):
    """
    Collect future information using the multi-step-look-ahead method to bias the
        movement of the next step

    Input: current position x = [[x1,y1],[x2,y2]...[xn,yn]],
           input_HP_sequence: the protein HP sequence input by user
           step_tmp: num of steps look-ahead in the algorithm
    Output: unormalized probabilities towards three different directions, i.e. left,
        right, and ahead

    """
    steps = min(steps_tmp,len(input_HP_sequence)-len(x))
    input_seq = input_HP_sequence[:(len(x)+steps)]

    couples = compute_couples(input_seq)

    # look ahead to get all possible configurations
    # do the first step
    [c1,d1] = one_step(x)
    # then do the remaining steps
    configs = c1
    dirs = d1 # directions for each configuration
    for s in range(steps-1):
        new_config = [] # a list of new configurations
        new_dirs = [] # a list of directions for new configurations
        for i,cf in enumerate(configs):
```

```
            [cfg,d] = one_step(cf)
            new_config += cfg
            new_dirs += [dirs[i]]*len(cfg)
        configs = new_config
        dirs = new_dirs

    # compute unnormalized probability for each configuration and find marginals
    # with respect to the direction
    [p_left, p_right, p_ahead] = [0.0,0.0,0.0]
    for i,cf in enumerate(configs):
        d = dirs[i]
        prob = np.exp(-compute_energy(cf,couples)/tau)
        if d == left:
            p_left += prob
        elif d == right:
            p_right += prob
        elif d == ahead:
            p_ahead += prob
    # return the unnormalized probabilities based on the immediate next step
    return [p_left, p_right, p_ahead]



def compute_next_point(pt1, pt2, move):
    """
    compute the next position of a newly added point
    """
    disp_vec = [pt1[0] - pt2[0], pt1[1] - pt2[1]]
    next_pt = [pt1[0], pt1[1]]

    if disp_vec == [0, 1]:
```

```
    if move == "left":
        next_pt[0] -= 1
    elif move == "right":
        next_pt[0] += 1
    elif move == "ahead":
        next_pt[1] += 1
    else:
        print "Unrecognized move direction!"
elif disp_vec == [0, -1]:
    if move == "left":
        next_pt[0] += 1
    elif move == "right":
        next_pt[0] -= 1
    elif move == "ahead":
        next_pt[1] -= 1
    else:
        print "Unrecognized move direction!"
elif disp_vec == [1, 0]:
    if move == "left":
        next_pt[1] += 1
    elif move == "right":
        next_pt[1] -= 1
    elif move == "ahead":
        next_pt[0] += 1
    else:
        print "Unrecognized move direction!"
elif disp_vec == [-1, 0]:
    if move == "left":
        next_pt[1] -= 1
    elif move == "right":
```

```
            next_pt[1] += 1
        elif move == "ahead":
            next_pt[0] -= 1
        else:
            print "Unrecognized move direction!"
    else:
        print "Wrong displacement bewteen two continuous points!"

    return next_pt


def resample_conformations(S, w, a, N_star):
    """
    perform standard resampling from the conformation set S and the probability vector a
    """
    assert(min(a) >= 0)

    # normalize the probability vector a
    a_normal = np.array(a)
    a_normal = a_normal / np.sum(a_normal)

    # resample conformations S_star with probabilities proportional to a_normal
    S_indice = np.arange(len(S))
    resample_obj = stats.rv_discrete(name = "resample_obj", values = (S_indice, a_normal))

    S_star_indice = resample_obj.rvs(size = N_star)

    S_star = []
    for i in S_star_indice:
        S_star.append(S[i])
```

```
    # update weights
    w_star = N_star * [1]

    # return the resampled conformations S_star and the weights w_star
    return (S_star, w_star)


def resample_with_pilot_exploration(S, w, input_HP_sequence, N_star):
    """
    Resample the current set of conformations using the pilot-exploration resampling
        scheme
    """
    a = [] # resampling probability vector
    # generate the next Delta residues m independent times for each conformation in S
    for j, x in enumerate(S):
        bj = 0
        for l in xrange(m):
            xl = list(x)
            pi_l = 0
            for i in xrange(Delta):
                # compute the probabilities along different directions
                [p_left, p_right, p_ahead] = multi_step_look_ahead(xl, input_HP_sequence,
                    rho)
                p_sum = p_left + p_right + p_ahead

                if p_sum == 0:
                    pi_l = 0
                    break
```

```
            # move a new step and update the weight of the conformation
            rand_num = random.random()
            if (rand_num < p_left/p_sum):
                next_pt = compute_next_point(xl[-1], xl[-2], "left")
                xl.append(next_pt)
                pi_l = p_left
            elif (rand_num < (p_left + p_right)/p_sum):
                next_pt = compute_next_point(xl[-1], xl[-2], "right")
                xl.append(next_pt)
                pi_l = p_right
            else:
                next_pt = compute_next_point(xl[-1], xl[-2], "ahead")
                xl.append(next_pt)
                pi_l = p_ahead

        # sum up the Boltzmann weight of each path l
        bj += pi_l # pi_l value exiting the loop

    # compute the unnormalized resampling probability aj
    bj /= m
    aj = bj**alpha
    a.append(aj)

# perform a standard resampling step with the probability vector
S_star, w_star = resample_conformations(S, w, a, N_star)

# return the resampled conformations S_star and the weights w_star
return (S_star, w_star)
```

```
def main():
    """
    Main function to implement the alogrithm SISPER
    """
    # start timer
    t1 = time.clock()

    global tau
    if len(sys.argv) == 4:
        sequence_file_name = str(sys.argv[1])
        output_file_name = str(sys.argv[2])
        tau = float(sys.argv[3]) # use tau as a global variable
    else:
        print "Missing inputs! Please provide: sequence_file_name, ouput_file_name, tau"

    # read in HP sequence
    input_HP_sequence = read_input_sequence(sequence_file_name)
    seq_len = len(input_HP_sequence)

    # connectivity couples
    couples = compute_couples(input_HP_sequence)

    # set random seed
    # random.seed(23)

    # initialization: fix the first step along the vertical direction
    S = []
    for i in xrange(N):
        S.append([[0, 0], [0, 1]]) # conformation set
    w = N * [1] # conformation weights
```

```
U = N * [0] # conformation energy

# sequentially generate conformations
for t in xrange(1, seq_len-1):

    # initialize a list to save the indice of incorrectly terminated conformations
    failed_indice = []

    # perform a regular SIS step with multi-step-look-ahead
    for i in xrange(len(S)):
        x = list(S[i])
        # compute the probabilities along different directions
        [p_left, p_right, p_ahead] = multi_step_look_ahead(x, input_HP_sequence, delta
            )
        p_sum = p_left + p_right + p_ahead

        # record the indices of incorrectly terminated conformations
        if p_sum == 0:
            failed_indice.append(i)
            continue

        p_left /= p_sum
        p_right /= p_sum
        p_ahead /= p_sum

        # move a new step and update the weight of the conformation
        rand_num = random.random()
        if rand_num < p_left:
            next_pt = compute_next_point(x[t], x[t-1], "left")
            x.append(next_pt)
```

```
            U_new = compute_energy(x, couples)
            w[i] *= np.exp(- (U_new - U[i]) / tau) / p_left
        elif (rand_num < (p_left + p_right)):
            next_pt = compute_next_point(x[t], x[t-1], "right")
            x.append(next_pt)
            U_new = compute_energy(x, couples)
            w[i] *= np.exp(- (U_new - U[i]) / tau) / p_right
        else:
            next_pt = compute_next_point(x[t], x[t-1], "ahead")
            x.append(next_pt)
            U_new = compute_energy(x, couples)
            w[i] *= np.exp(- (U_new - U[i]) / tau) / p_ahead

        # save the energy of the current configuration
        U[i] = U_new

        # update S
        S[i] = list(x)

    # remove incorrectly terminated conformations
    for i, index in enumerate(failed_indice):
        del S[index - i] # shift the index due to deletion; work for the sorted list
        del w[index - i]
        del U[index - i]

    # perform resampling
    if t % (lamb + 1) == 0:
        S, w = resample_with_pilot_exploration(S, w, input_HP_sequence, len(S))
        # update the conformation energies
        for i, x in enumerate(S):
```

```
                U[i] = compute_energy(x, couples)

        # print progress
        print 'Progress %d/%d' % (t, seq_len-2)

    # record execution time
    t2 = time.clock()
    t_exec = t2-t1

    # save conformations and execution time
    write_conformations(S, w, U, t_exec, output_file_name)

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

2. plot_conformations.py

```
# STAT5252-SISPER
#
# This script is used to plot protein conformations.
#
# Usage:
# python plot_conformations.py sequence_file_name conformation_file_name
    fig_file_keywords num_of_figs


import matplotlib.pyplot as plt
import matplotlib.ticker as tk
import numpy as np
import sys


def read_input_sequence(sequence_file_name):
    """
        read in the input HP sequence from file
        """
    input_HP_sequence = ''
    with open(sequence_file_name, 'r') as file_id:
        for line in file_id:
            input_HP_sequence += line.replace('\n', '')
    return input_HP_sequence


def read_conformations(conformation_file_name):
    """
```

```
    read from file the conformations sorted by energy from low to high
    """
    conformations = np.load(conformation_file_name)
    U = conformations['energies'].tolist()
    S = conformations['coordinates'].tolist()
    w = conformations['weights'].tolist()
    return (U, S, w)



def plot_config(x_coord,input_HP_sequence,title,figname,is_grid_plotted):
    """
    Input: x_coord: coordinates of all the protein residues = [[x1,y1],[x2,y2]...[xn,yn]]
           input_HP_sequence: HP sequence of protein (e.g. 'HPPHHP')
           title: figure title
           figname: figure name that will be saved as
           is_grid_plotted: whether to plot grids = True/False.
    """
    plt.figure(1)
    plt.clf()
    # figure parameters
    dot_size = 30
    line_wid = 1
    # first plot all individual residues
    for i,coord in enumerate(x_coord):
        HP = input_HP_sequence[i]
        if HP == 'H':
            fill_col = 'k' # filled color of the dot is black for 'H' residue
        elif HP == 'P':
            fill_col = 'none' # no fill for 'P' residue
        plt.scatter(coord[0],coord[1],s = dot_size, facecolor = fill_col, edgecolor = 'k')
```

```
    # then plot connected lines between adjacent residues
    x_coord_np = np.array(x_coord) # convert to numpy array
    plt.plot(x_coord_np[:,0],x_coord_np[:,1],'k-',linewidth=line_wid)
    # add labels and title
    plt.title(title)
    if is_grid_plotted:
        plt.xlabel('x')
        plt.ylabel('y')
        plt.grid(True)
        # only show tickers at integers
        plt.gca().axes.get_yaxis().set_major_locator(tk.MaxNLocator(integer=True))
        plt.gca().axes.get_xaxis().set_major_locator(tk.MaxNLocator(integer=True))
    else:
        # make both axes invisible
        plt.gca().axes.get_xaxis().set_visible(False)
        plt.gca().axes.get_yaxis().set_visible(False)
        # remove the frame
        plt.gca().set_frame_on(False)
    # set aspect ratios to be equal
    plt.gca().axes.set_aspect('equal')
    # save figure
    plt.savefig(figname)



# --- main code ---
def main():
    if len(sys.argv) == 5:
        sequence_file_name = sys.argv[1]
        conformation_file_name = sys.argv[2]
        fig_file_keywords = sys.argv[3]
```

```
        num_of_figs = int(sys.argv[4])
    else:
        "Error in input! Please provide: sequence_file_name, conformation_file_bame,
            num_of plots"

    # read in data
    input_HP_sequence = read_input_sequence(sequence_file_name)
    U, S, w = read_conformations(conformation_file_name)

    # plot conformations
    seq_len = len(input_HP_sequence)
    assert(num_of_figs <= len(S))

    for i in xrange(num_of_figs):
        title = "conformation: length = %d, energy = %g" % (seq_len, U[i])
        fig_name = fig_file_keywords + "_%d.pdf" % (i+1)
        plot_config(S[i], input_HP_sequence, title, fig_name, True)


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