Computational Physics - Protein folding of a lattice protein using the metropolis Monte Carlo algorithm

H.F. Stegenga - S3860205

November 2020

1 Introduction

1.1 Protein folding

Protein folding is a process that occurs in nature in which a protein acquires it's native structure. The structure in which the protein is folded depends on the sequence of monomers inside it. In this assignment the Metropolis Monte Carlo algorithm is used to simulate a simple protein model consisting of 2 types of monomers. Either H or P monomers. The goal is to find the lowest energy state possible by simulating the process of protein folding at various temperatures.

1.2 Metropolis Monte Carlo

Metropolis Monte Carlo is an algorithm in where the protein is randomly perturbed in order to attempt to reduce it's energy level. This energy level is dependent on how many H monomers are next to each other. More neighbouring H monomers means lower energy levels. The algorithm simulates protein folding by randomly perturbing the protein chain and accepting changes which lower the energy level, in an attempt to eventually find the lowest energy state possible. In order to allow some control over how local minimums in the conformation are handled, a temperature parameter is used which controls the probability of higher energy state moves being accepted.

1.3 Simulated Annealing

In the later parts of the assignment the temperature parameter is gradually reduced. This is called Simulated Annealing. The idea is to more efficiently search through the solution space of moves (random perturbations) in an attempt to find the global minimum energy level. This is done by gradually lowering the temperature, and thus the probability of accepting bad moves reduces as temperature decreases. The rate at which this is done is called the cooling schedule. This allows the algorithm to accept bad moves when necessary in order to get out of local minimums more efficiently than MMC does, and therefore the likelihood of finding the global minimum is higher. While Simulated Annealing is a rather general and often used approach for various optimization problems, specifically for HP proteins there are various approaches available which exploit the specifics of the protein structure in order to more efficiently find the global minimum [1].

1.4 Code style

A set of practices that are not taught as part of the course, or diverge from the practices introduced in the course or assignment instruction, are implemented as part of the source code. For example, in order to improve code correctness and readability, type hinting has been introduced. Type hinting decreases the chance of incorrect datatypes being used, preventing subtle bugs. In a similar fashion, where necessary, explicit numeric casts have been implemented in order to guarantee correct computations.
Functions which are supposed to use the random number generator are modified such that the random numbers required are part of the function signature instead. For example, the *perform_pivot_move* function relies on random rotation, this is instead passed as a parameter, rather than generated as part of the function implementation. This therefore transforms the functions into pure functions: they only rely on input arguments, and produces an identical output given an identical input. This makes the function predictable and testable, and thus easier to verify correctness. The random numbers are therefore purely generated inside the *mmc* function.

[1] https://www.brics.dk/RS/99/16/BRICS-RS-99-16.pdf

2 Implementation

2.1 Preparing, visualizing and analyzing a chain

2.1.1 Writing a (self-avoiding) random walk function

For the random walk algorithm I decided to implement a simple algorithm which takes discrete steps in either of 4 directions. Initially I made a very simple random walk algorithm. However, it turned out to often get stuck after a small number of moves. Therefore, I needed to add the constraint of it being self-avoiding. Which means it cannot intersect with itself or be overlapping. This adds complexity to the randomwalk algorithm, since it means that we need to detect situations in which the random walk gets stuck. As such, I have implemented a backtracking mechanism into the algorithm. In case the random walk cannot progress, it retraces it's steps until it isn't stuck anymore, excluding the taken path for the future. This guarantees us that it will always, however long it might take, converge to a solution[2].

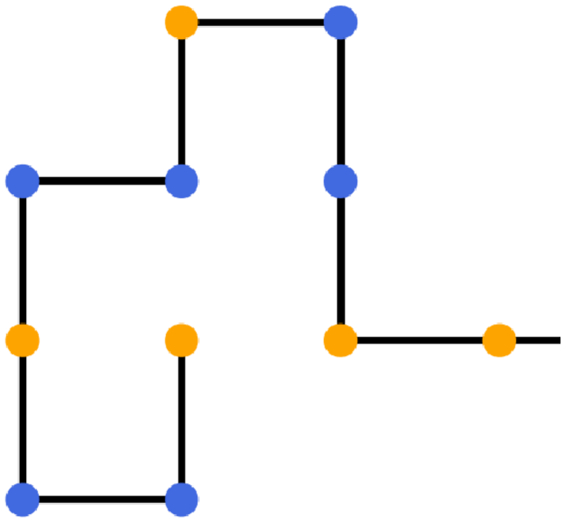


Figure 1: An example where the randomwalk get's stuck and needs to retrace it's steps.

The next page contains the pseudo-code of the implemented randomwalk algorithm. The final implementation of the algorithm in Python can be found in the submitted files.

[2]Unless the random generator is broken

```
# N is length of the desired protein, H_chance is a chance 0-1
# whether it's a H or P monomer.
func random_walk(N, H_chance):

    # To track excluded nodes
    # This is to exclude nodes in case we get stuck and need to backtrack
    excluded = []

    # Initialize the chain with a initial new node at (0,0)
    chain = [create_node(kind: choose_monomer_kind(H_chance), (0, 0)]

    loop:
        # Randomly walk in a direction, up/down/left/right
        direction = choose_direction()

        # Compute the new coordinate based on last node in the chain and direction
        new_coord = compute_last_coord(current_chain.last(), direction)

        # Evaluate a function which checks if the node can be placed.
        # This function not only checks for neighbours, but also checks the excluded list.
        if is_valid_new_position(new_coord, current, excluded):

            chain.append(create_node(kind: choose_monomer(H_chance), new_coord))

        # In case we cannot place the node, because it's either
        # blocked or excluded because it's a dead end
        else:

            # Append the new node coord to the excluded list
            excluded.append(new_coord)

            # We now need to check if the previous node, i.e. last one
            # in the chain, is a dead end.
            # This can occur if the new coord was the last available direction,
            # in which case we must go back, as there is no way to continue
            # In such a situation we need to remove the last node and
            # add it to the excluded list, therefore the algorithm continues
            # again at the node before the previous one. (Where new_coord is the 'current' one)
            # Then the algorithm explores new directions which where not yet excluded/visited.

            previous = chain.last()
            if is_dead(previous):
                chain.pop()
                excluded.append(previous)

        # The final thing we need to do at each iteration in our loop is to
        # check whether we have reached a chain of N nodes
        # Since our algorithm includes backtracking it will always converge to a solution
        if N == chain.len():
            return chain
```

Figure 2: Pseudo code for the Self-Avoiding Random Walk function.

2.1.2 Checking weighting factor

In order to check whether our randomwalk function works correctly, a couple of small functions have been made which allow to print the protein chain and check the hydrophobic fraction. A possible output for the hydrophobic fraction check function has been shown below in figure 3. Overrides for the string print function have been made as well for the monomer class, allowing me to print out all the monomers with their type and coordinates so that I can verify correctness. An example output of this can be seen in figure 4.

```
0.48 12/13 - HHPPPHHPPHPPHHHPPPHPPHPHH
0.56 14/11 - PPHHPPHPHHPHPHHHPHHHHPHPP
0.52 13/12 - PHPHHPHPHPPPHHHHPHPHPPPHH
0.4 10/15 - HPPPHPPPHPPHHHPPHPPPPPHHH
0.6 15/10 - HHHHPPHPHHPHHPPPHHPPHHHHP
0.36 9/16 - HPPHPHPPPHHHPPPPPHPPPPHPH
0.44 11/14 - HHPPPPPPHHPPHPPHHPPPHPHHH
0.64 16/9 - HPHHPHHHPHHHPHPHPHHPPHHPH
0.52 13/12 - HPHHHPPHPPPPHPPHHPHHHHHPP
0.56 14/11 - PHHHHPPPPPHHPPHPHHHPHPHHH
0.68 17/8 - HHHHPPPPHPHHHPHPHHHHHHHPH
0.56 14/11 - HHHPHPPHPHHHPPPPHHHPHPPHH
0.52 13/12 - HPPHPPPHHPPPPHHHPHHHHHHPP
0.52 13/12 - HPPPPHHHPPPHPHHHHHHPHHPPP
0.56 14/11 - HHPHHPPHHHHHHPPPHHPPPPPHH
0.48 12/13 - HHPHHPPHPPPHPPHHHPHPPPHHP
0.6 15/10 - PPHHPPHPPHHHHPPHHHHHPHHPH
0.4 10/15 - PPPPPPPPPHHHPHHHPPPHHHHPP
0.52 13/12 - PPHHPHPHPPPHHPHHPHHPHHPPH
0.36 9/16 - HPPPPPPPHPHPHPPHPPPPHHPHH
0.52 13/12 - PPHPHPHPPPHHPHPHHPPHHPHHH
0.36 9/16 - PPHPHHPHPPPPHPHHPHHPPPPPP
0.48 12/13 - PPPHPHPHHPHPPHPPHPHHPHHHP
0.48 12/13 - PHHHPPHHPPPHPPHHHPPPPHHHP
0.4 10/15 - PPPHHPPPPHHHPPPPHPHPHPHHP
Average hydrophobicity: 0.50
```

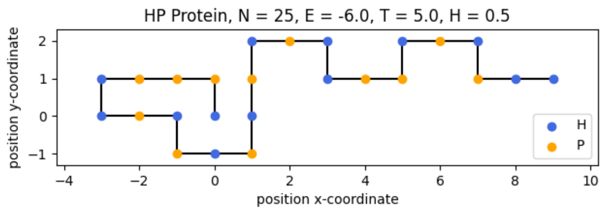
Figure 3: Using a hydrophobicity of 0.5, yields an average of 0.5 (or close to it)

```
Monomer hydrophobic fraction, count of H, count of P and composition order:
0.56 14/11 - PPPHHPPHHPHPHHPHPPHHHHHPH

Generated coordinates and monomer types of the above monomer chain:
[((0,0) P), ((0,1) P), ((-1,1) P), ((-1,2) H), ((-1,3) H), ((-1,4) P), ((-2,4) P), ((-2,5) H),
((-2,6) H), ((-2,7) P), ((-3,7) H), ((-3,8) P), ((-4,8) H), ((-5,8) H), ((-5,9) P), ((-6,9) H),
((-6,10) P), ((-6,11) P), ((-7,11) H), ((-7,12) H), ((-6,12) H), ((-6,13) H), ((-7,13) H),
((-8,13) P), ((-8,14) H)]
```

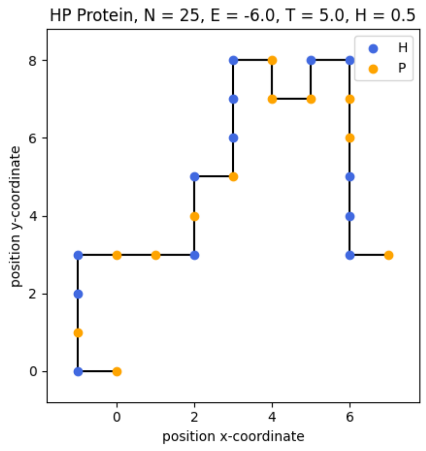
Figure 4: Example output of the monomer string print function.

2.1.3 Writing a visualization function.

A relatively simple drawing function has been implemented using the matplotlib library. Using these drawings we can visually verify whether moves are correctly implemented, as well count the HH encounters. It also shows the length N, energy E, temperature T and hydrophobicity H.



(a)



(b)

Figure 5: Examples of two generated visualizations of monomer chains.

```
def plot_protein(lattice: ProteinLattice, temperature: float, hydrophobicity: float):
    plt.title('HP Protein, N = {}, E = {}, T = {}, H = {:.2}'.format(
        len(lattice.chain),
        calculate_energy(1.0, lattice),
        temperature,
        hydrophobicity
    ))

    plt.plot([elem.x for elem in lattice.chain],
             [elem.y for elem in lattice.chain],
             'k-', zorder=0)

    plt.scatter([elem.x for elem in lattice.chain if elem.kind == MonomerKind.H],
                [elem.y for elem in lattice.chain if elem.kind == MonomerKind.H],
                zorder=3, label='H', color=blue)

    plt.scatter([elem.x for elem in lattice.chain if elem.kind == MonomerKind.P],
                [elem.y for elem in lattice.chain if elem.kind == MonomerKind.P],
                zorder=3, label='P', color=orange)

    plt.margins(0.1)
    plt.ylabel('position y-coordinate')
    plt.xlabel('position x-coordinate')

    plt.legend()
    plt.gca().set_aspect('equal')
    plt.show()
```

Figure 6: Python code used for visualization of monomer chains.

2.1.4 Writing the energy level computation function.

The implemented energy level calculation is relatively simple. It does however take advantage of storing the chain in 2 data-structures in order to significantly speed up the time it takes to compute the energy level. The lattice data-structure not only keeps an ordered array of monomers, but also stores a map which allows for lookups by coordinate. This means we don't need to iterate over the whole chain of monomers to detect neighbours, but can take a fixed amount of 4 samples to gather all neighbouring monomers. This significantly speeds up computation for the energy level function, effectively reducing complexity of the computation from $O(N^2)$ to $O(4N)$, where N is the length of the monomerchain. Next to this, the dual data-structure approach maximizes cache efficiency (by keeping a small and constant sized array and constant sized map) and limits memory consumption as well. Various helper functions are implemented as well in said data-structure.

```
# Returns the energy level of the chain
# Epsilon is the energy associated with two H contacts
func calculate_energy(epsilon, lattice):
         # Initially zero encounters
         f = 0

         # Iterate over all monomers in the chain
         for monomer in lattice.chain:

                 # We are only interested in H monomers
                 if monomer != H:
                         continue

                 # Iterate over all neighbours
                 for neighbour in lattice.get_neighbours(monomer):
                         # Again, we're only interested in H monomers
                         if neighbour != H:
                                 continue

                         # Add 1 to f
                         f += 1


         # Effectively we walk the chain twice, thus counting all contacts twice.
         # Therefore we have to simply divide by 2 to obtain f.
         # Because we count them exactly twice, we can use integer mathematics as f is
         # guaranteed to be divisible by 2
         f = f / 2

         return -1.0 * epsilon * f
```

Figure 7: Pseudo code for the energy level calculation function.

2.2 Implementing the Monte Carlo moves

2.2.1 Kink jump and endpoint rotation

Both the kink jump and endpoint rotate moves are implemented in a unconventional but very similar manner. The most straightforward implementation of either of the moves would be to compute a difference, check it using the lattice, and if not successfully continue or else return. This would have to be done for up to 12 computations + if statements in total across both moves. This would not only significantly affect (cognitive) function complexity, but also destroy performance due to excessive unpredictable branching. (Branching is relatively expensive compared to simple computations). To mitigate this, a set of lookup tables are used which completely eliminate all branching related to checking all the various possibilities of neighbours. Instead, based on a single computation of the positional differences between the previous/next and current monomer, at most 2 branches are checked in the lattice. These checks again have a complexity of O(1) due to our dual data-structure approach as described earlier, further increasing performance as compared to a naive approach. The basic structure of the kink jump / endpoint rotate move function is therefore really simple:

```
# Tries to attempt a kink jump, returns true if success or not.
# jump_idx is an index of the rotate/jump monomer in the chain.
func kink_jump(jump_idx, lattice):
    # Check if we are at an endpoint in our monomer chain
    if jump_idx == 0 or lattice.chain.len() - 1 == jump_idx:
        # Rotate endpoint

        # 1. Compute diffs in position between prev and current monomer.
        # 2. Use lookup table to get possible new positions.
        # 3. Check new positions and move if possible.

    else:
        # Kink jump

        # 1. Compute diffs in position between prev, current and next monomer.
        # 2. Use lookup table to get possible new positions.
        # 3. Check new positions and move if possible.

    # None of the offsets were free, returning False.
    # Either of the cases above will automatically end up
    # here if no positions were available.
    return False
```

Figure 8: Pseudo code structure of the kink jump / endpoint rotate function

As can be seen in listing 9 and 11, the code in either case of the move is very similar. Any difference is mostly in the computation of the positional differences. In the case of the endpoint rotation only the previous monomer has to be considered, whereas in the case of the kink jump both previous and next monomers are considered. The lookup tables are slightly different between them, since the lookup table for kink jumps needs to take the orientation into consideration as well. These can be found, as Python code, in listing 10 and 12.

```
# Rotate endpoint

# Determine what the previous monomer is.
if jump_idx == 0
    # Second monomer
    prev_monomer = lattice.chain[1]
else
    # Pre-last monomer
    prev_monomer = lattice.chain[-1]

# Determine the positions to test for a possible rotation based
# on the difference in coordinates using a lookup table.
offsets = endpoint_rotate_lookup(
                prev_monomer.x - monomer.x,
                prev_monomer.y - monomer.y)

# We have at most 2 offsets we need to check.
for (x,y) in offsets
    # Check if the offset is free.
    if not lattice.has_monomer(prev_monomer.x + x,
                               prev_monomer.y + y)
        # Move the monomer in the lattice/chain.
        lattice.move_monomer(jump_idx,
                             prev_monomer.x + x,
                             prev_monomer.y + y)
        return True
```

Figure 9: Pseudo code for endpoint rotate case

```
# Returns the positions to check for a diff between previous and current points.
# Specifically for endpoint rotations.
def endpoints_rotate_lookup_table(diff_x: int, diff_y: int) -> List[Tuple[int, int]]:
    return {
        (0, +1): [(1, 0), (-1, 0)],  # If prev is above, return right/left
        (0, -1): [(1, 0), (-1, 0)],  # If prev is below, return right/left
        (+1, 0): [(0, 1), (0, -1)],  # If prev is right, return top/bottom
        (-1, 0): [(0, 1), (0, -1)],  # If prev is left, return top/bottom
    }[(diff_x, diff_y)]
```

Figure 10: Python code for the lookup tables for endpoint rotations

```
# Kink jump

# Determine prev and next monomers in the chain.
prev_monomer = lattice.chain[jump_idx - 1]
next_monomer = lattice.chain[jump_idx + 1]

# Determine the positions to test for a possible kink jump,
# based on difference in coordinates using a lookup table.
offsets = kink_jump_lookup_table(prev_mon.x - monomer.x,
                                 prev_mon.y - monomer.y,
                                 next_mon.x - monomer.x,
                                 next_mon.y - monomer.y)

# We have at most 2 offsets we need to check.
for (x,y) in offsets
    # Check if the offset is free.
    if not lattice.has_monomer(prev_monomer.x + x,
                               prev_monomer.y + y)
        # Move the monomer in the lattice/chain.
        lattice.move_monomer(jump_idx,
                             prev_monomer.x + x,
                             prev_monomer.y + y)
        return True
```

Figure 11: Pseudo code for kink jump case

```python
# Returns position of potential kink jump, if any, based on neighbour positions.
# Returns List of coordinates with len = 1, or len = 0 if no potential kink jump.
def kink_jump_lookup_table(diff_prev_x: int,
                           diff_prev_y: int,
                           diff_next_x: int,
                           diff_next_y: int) -> List[Tuple[int, int]]:
    return {  # Only eight possible configurations can match, otherwise return empty list
        # Clock Wise
        ((0, 1), (1, 0)): [(1, 1)],  # prev above, next right
        ((1, 0), (0, -1)): [(1, -1)],  # prev right, next bottom
        ((0, -1), (-1, 0)): [(-1, -1)],  # prev bottom, next left
        ((-1, 0), (0, 1)): [(-1, 1)],  # prev left, next top

        # Counter Clock Wise
        ((0, 1), (-1, 0)): [(-1, 1)],  # prev above, next left
        ((-1, 0), (0, -1)): [(-1, -1)],  # prev left, next bottom
        ((0, -1), (1, 0)): [(1, -1)],  # prev bottom, next right
        ((1, 0), (0, 1)): [(1, 1)],  # prev right, next top
    }.get(((diff_prev_x, diff_prev_y), (diff_next_x, diff_next_y)), [])
```

Figure 12: Python code for the lookup tables for kink jumps

2.2.2 Pivot move

The pivot move is implemented as a simple procedural function. Again, checks are implemented as O(1) operations, as such the whole function basically has a computational complexity increasing only as a function of the rotated part length. A future potential optimization would be to do an early lattice check directly after computing each rotated monomer, saving computations, but this would require a faster way to exclude those from the rotated part. If you would implement this as a simple filter, it would become a $O(N-1)$ operation in worst case at every step, so it isn't worth it. Therefore, we do a simple pass over the rotated part a second time after applying 1 filter to it. Given our small chain sizes, any clever optimization here would likely have minimal effect, hence I have chosen not to implement such an optimization.

```
func perform_pivot(rotation_point_idx, direction, rotated_side, lattice)
    # The monomer at the rotation point.
    rotation_monomer = lattice.chain[rotation_point_idx]

    # The monomers that are being rotated
    rotated_part = gather_rotated_monomers(lattice, rotation_point_idx, rotated_side)

    # Get the rotation angle (Either 3 pi/2 or pi/2)
    angle = get_angle(direction)

    new_positions = []
    for monomer in rotated_part
        # Translate to 0,0 relative to rotation_monomer.
        shifted_x = monomer.x - rotation_monomer.x
        shifted_y = monomer.y - rotation_monomer.y

        # Rotate the monomer
        a1 = shifted_x * cos(angle)
        a2 = shifted_y * sin(angle)
        a3 = shifted_x * sin(angle)
        a4 = shifted_y * cos(angle)

        rotated_shifted_x = int(round(a1 - a2))
        rotated_shifted_y = int(round(a3 + a4))

        # Translate the monomer back to original position and add it to new_positions
        new_positions.push((rotated_shifted_x + rotation_monomer.x,
                           rotated_shifted_y + rotation_monomer.y))

    # Check if all coordinates are free
    # We can exclude those that are rotated to positions
    # in the original configuration, those are always free
    for x,y in new_positions.filter(rotated_part)
        if lattice.has_monomer(x, y)
            return False

    # Move all monomers to new positions
    lattice.move_monomers(rotated_part, new_positions)

    return True
```

Figure 13: Pseudo code for pivot move function

2.3 Full MMC program

The full MMC algorithm is implemented in the *mmc* function, of which the pseudo code can be found in listing 14. The algorithm diverges slightly from the prescribed variant, since all random numbers are also generated in this function. This way, the individual functions responsible for the moves can be implemented as pure functions without (almost any) side-effects. The only side-effect is that those functions might modify the lattice which is given. The primary reason for this is for efficiency and performance. While copying the lattice at every step and reverting to the old one would simplify the algorithm and implementation, it is very bad for performance as well. This is due to the relatively slow speed at which new memory can be allocated, which would dominate the runtime of the algorithm and slow it down significantly. Instead, an undo stack is implemented in the lattice class. This records the last taken move such that they can be undone if the move is not accepted. This way any memory allocations in the primary loop of the program are avoided, while still keeping the algorithm conceptually relatively simple. (Obviously a few initial allocations need to be made in the first steps of the simulation, but these can never exceed a constant amount dependent on the chain length. As such, if such allocations occur, they will in the first few steps.)

The actual implementation also sets up the random number generator with a fixed input such that we always start with the same initial protein configuration. After the protein has been generated, a new random seed is set and simulation starts. By using this technique to set the initial configuration, no unnecessary loading and saving of data from and to disk is required, and generation of such relatively small length protein chains is way faster than loading data from disk. (Although obviously not when very large chain sizes are used.)

Another small change which is not detailed in the pseudocode is that I modified the MMC function such that it cannot become stuck. In rare cases it's possible that no kink jump or endpoint rotation move can be performed, in which case the mmc function gets stuck in an infinite loop. In order to fix this it tracks the already attempted kinkjump positions, and if none is available a pivot move is performed instead. Since in such situations the only way to get 'unstuck' is to perform a pivot move. This immediately is also a small optimization, since it will exclude all positions for which it already attempted a kink jump in the same mmc iteration. This saves some computation time since often kinkjumps and end point rotations can only be performed at a rather limited amount of positions relative to the protein length.

```
# Implements the full MMC protein folding algorithm
func mmc(length, hydrophobicity, max_iterations, temperature, epsilon = 1, boltzmann = 1)

    samples = []
    # Generate protein and calculate base energy level.
    lattice = generate_lattice(length, hydrophobicity)
    energy = calculate_energy(epsilon, lattice)
    # Add initial energy level
    samples.append(energy)

    for iteration in range(0, max_iterations)
        # Pick the move to performs
        move_kind = random([0,1])
        if move_kind = 0
            # kink jump
            success = false
            while not success
                # pick monomer
                jump_idx = random([chain])
                success = perform_kink_jump(jump_idx, lattice)
        else
            # pivot
            success = false
            while not success
                rotation_idx = random([chain])
                direction = random([0,1])
                side = random([0,1])
                success = perform_pivot(rotation_idx, direction, side, lattices)

        # calculate new energy level after performing a move
        new_energy = calculate_energy(epsilon, lattice)
        if new_energy < energy
            energy = new_energy
        else
            # Compute boltzmann weight and either accept or reject move
            w = exp(- (new_energy - energy) / (boltzmann * temperature))
            if w > random()
                energy = new_energy
            else
                lattice.undo_last_move()

        # Sample energy every so often
        if iteration+1 % sampling_frequency == 0
            samples.append(energy)

    plot_protein(lattice)
    return samples
```

Figure 14: Pseudo code for MMC function.

3 Results and discussion

3.1 Benchmarking

3.1.1

Before benchmarking can begin tests were performed to get an idea of the input parameters. For example, we need to set the temperature and mmc iterations parameters such that (some form) of convergence is reached. The following HP sequence will be used for the benchmarking procedure along with a reduced temperature parameter of 0.25, sampling frequency of 100, 50.000 mmc iterations, moving average of 3 and a hydrophobic fractions of 0.2, 0.5 and 0.8. The structure of the proteins is identical, only the specific monomer types differ between the different hydrophobic fractions.

```
Monomer hydrophobic fraction, count of H, count of P and composition order:
0.56 14/11 - PHHPHHPHHPPHPHPHHHPPHHPHP

Generated coordinates and monomer types of the above monomer chain:
[((0,0) P), ((0,-1) H), ((-1,-1) H), ((-1,0) P), ((-1,1) H), ((0,1) H),
((0,2) P), ((1,2) H), ((1,1) H), ((1,0) P), ((2,0) P), ((3,0) H), ((3,1) P),
((3,2) H), ((4,2) P), ((4,1) H), ((4,0) H), ((5,0) H), ((5,1) P), ((6,1) P),
((6,0) H), ((7,0) H), ((8,0) P), ((8,-1) H), ((8,-2) P)]
```

Figure 15: The used protein for the benchmarking procedure (at H = 0.5)

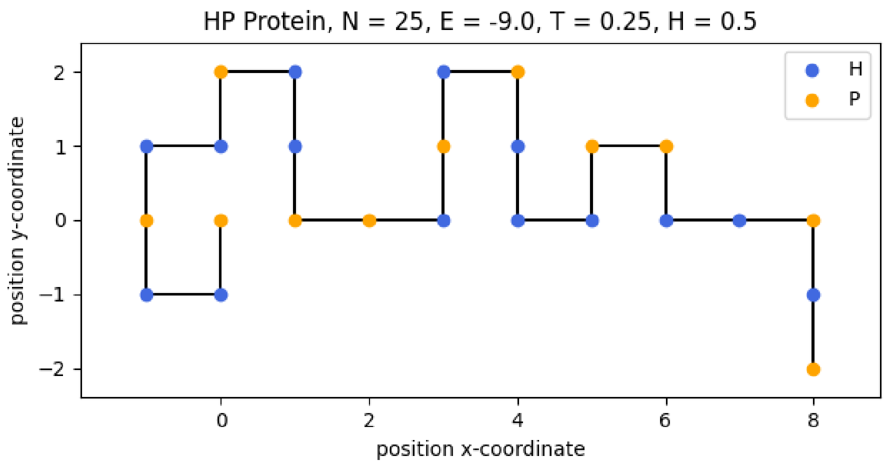


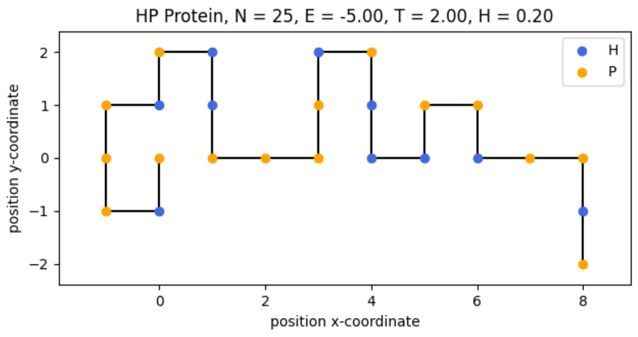
Figure 16: Initial configuration as used in our benchmark simulations (at H = 0.5)

```
Protein H = 0.2

[((0,0) P), ((0,-1) H), ((-1,-1) P), ((-1,0) P),
((-1,1) P), ((0,1) H),((0,2) P), ((1,2) H),
((1,1) H),((1,0) P), ((2,0) P), ((3,0) P),
((3,1) P), ((3,2) H), ((4,2) P), ((4,1) H),
((4,0) H), ((5,0) H), ((5,1) P), ((6,1) P),
((6,0) H), ((7,0) P), ((8,0) P), ((8,-1) H),
((8,-2) P)]
```

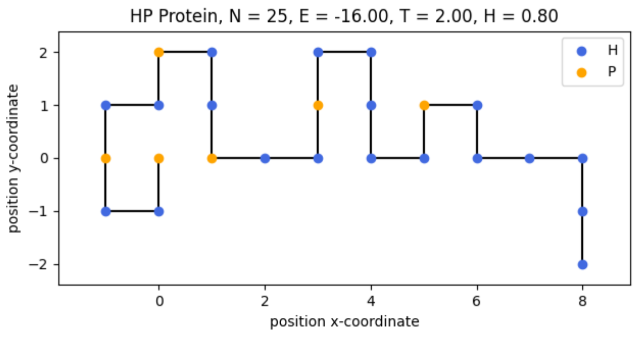
(a)

```
Protein H = 0.8

[((0,0) P), ((0,-1) H), ((-1,-1) H), ((-1,0) P),
((-1,1) H), ((0,1) H), ((0,2) P), ((1,2) H),
((1,1) H), ((1,0) P), ((2,0) H), ((3,0) H),
((3,1) P), ((3,2) H), ((4,2) H), ((4,1) H),
((4,0) H), ((5,0) H), ((5,1) P), ((6,1) H),
((6,0) H), ((7,0) H), ((8,0) H), ((8,-1) H),
((8,-2) H)]
```

(b)



(c)

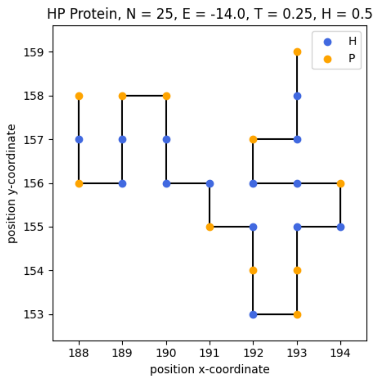


(d)

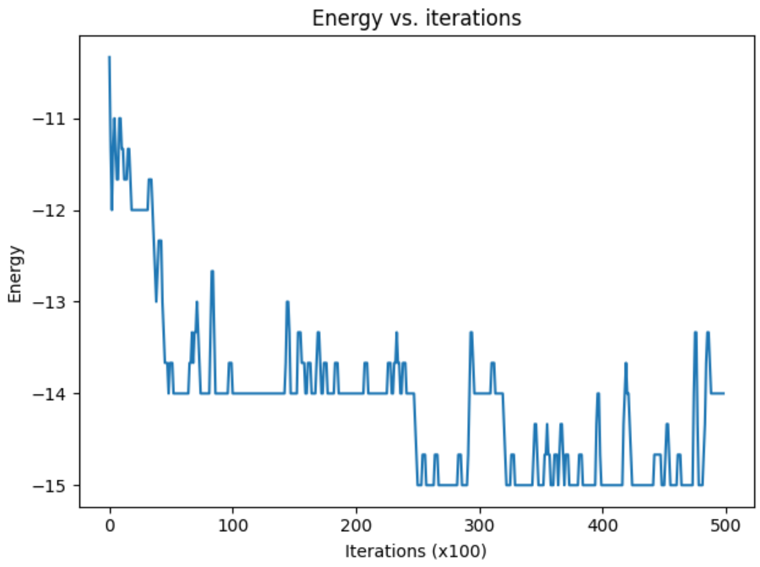
Figure 17: Resulting protein conformations for H = 0.2 and H = 0.8

3.1.2

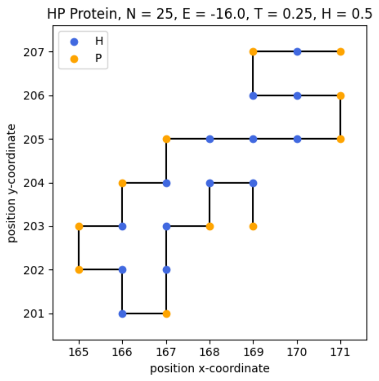
Question a and b are answered using the following configurations and graphs.



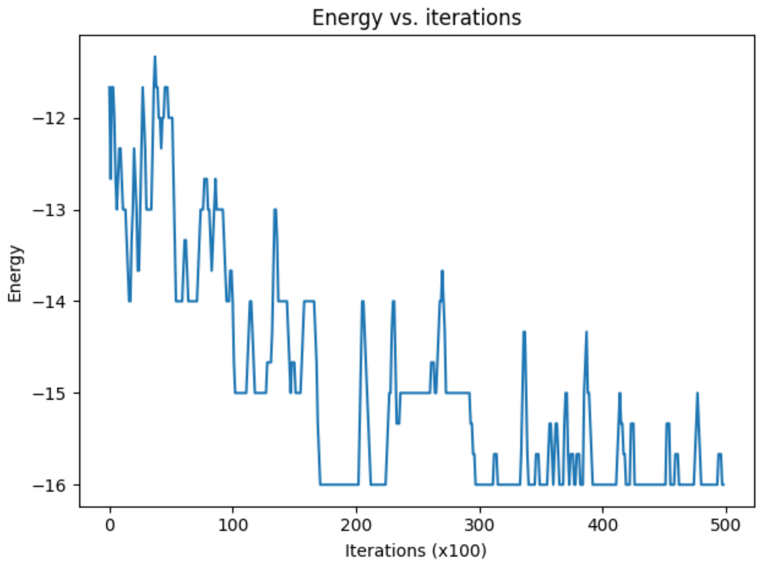
(a)



(b)

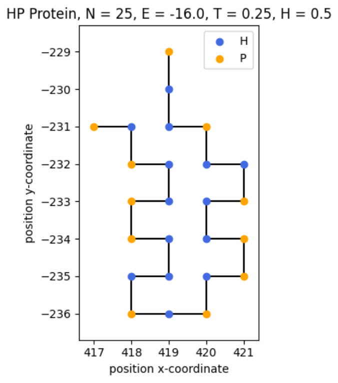


(c)

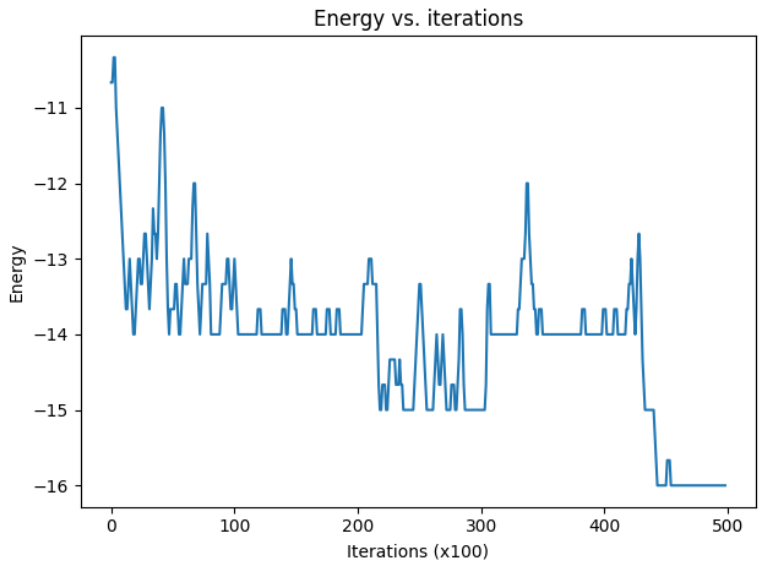


(d)

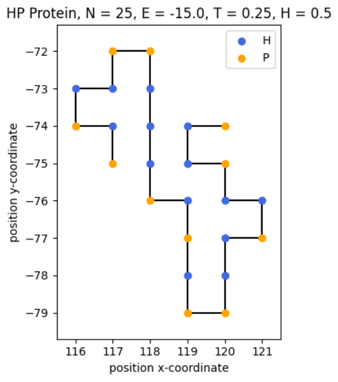
Figure 18: Examples of energy vs. iterations (T = 0.25, H = 0.5, 50K iterations, moving average = 3)



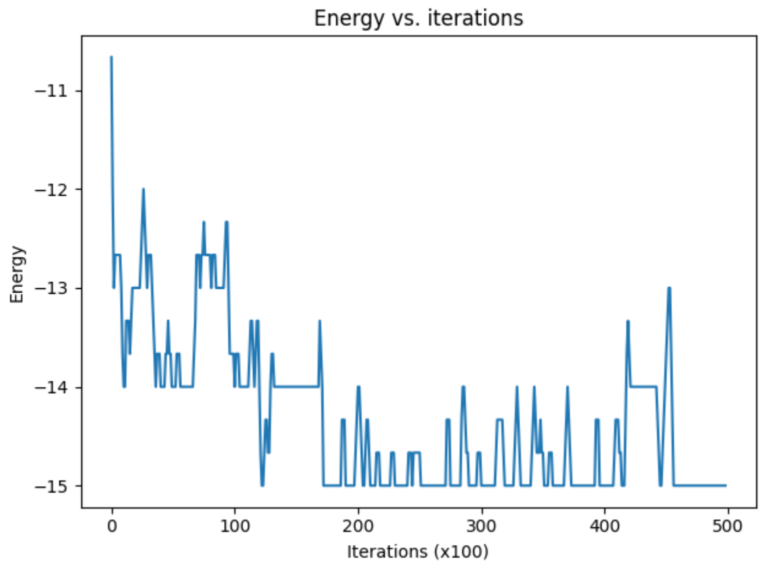
(a)



(b)



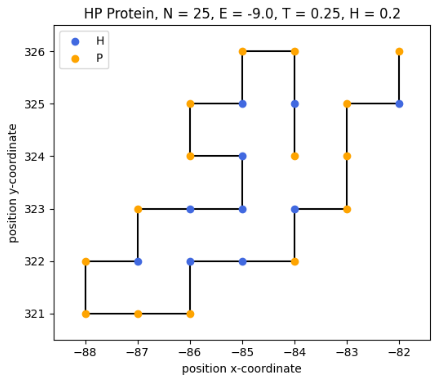
(c)



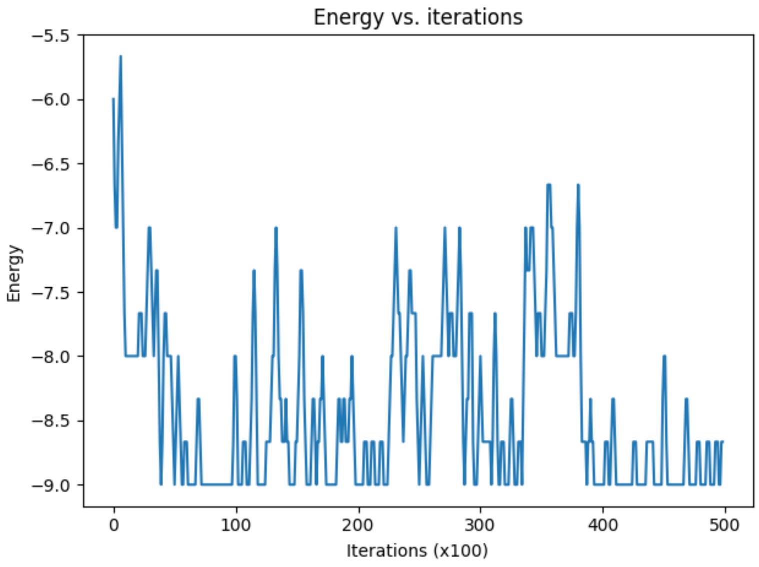
(d)

Figure 19: Examples of energy vs. iterations (T = 0.25, H = 0.5, 50K iterations, moving average = 3)

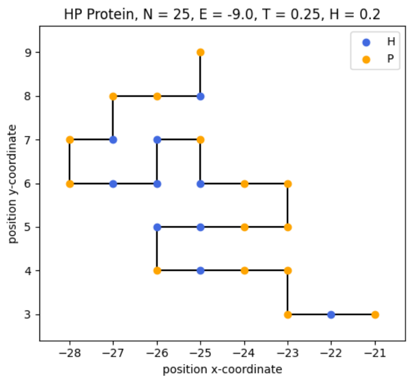
Question c is answered using the following configurations and graphs.



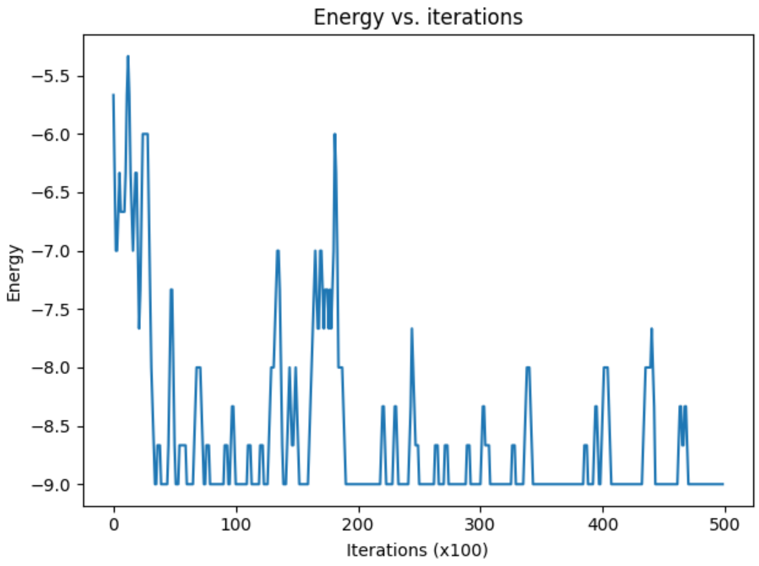
(a)



(b)

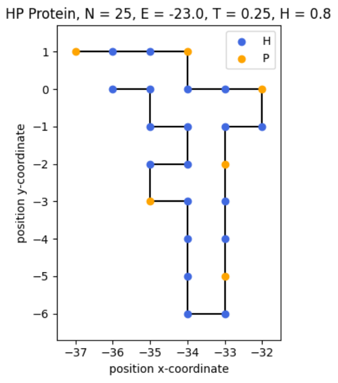


(c)

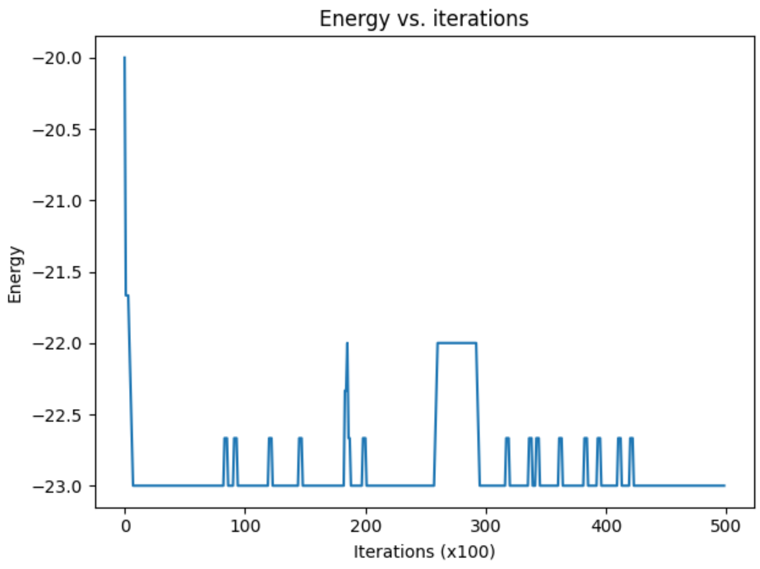


(d)

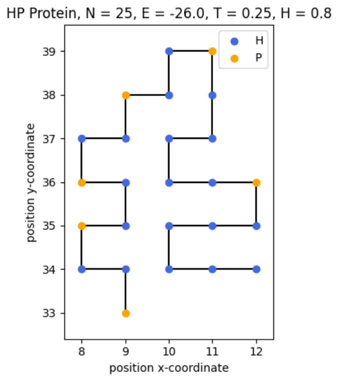
Figure 20: Examples of energy vs. iterations (T = 0.25, H = 0.2, 50K iterations, moving average = 3)



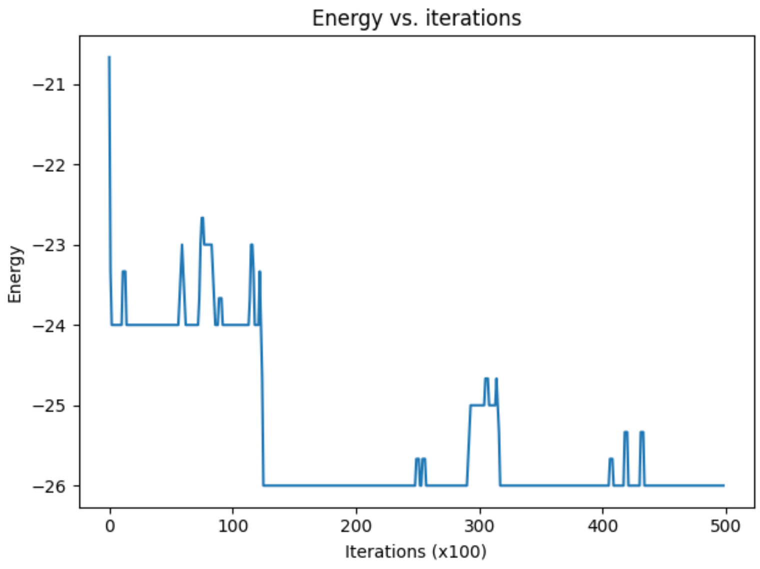
(a)



(b)



(c)



(d)

Figure 21: Examples of energy vs. iterations (T = 0.25, H = 0.8, 50K iterations, moving average = 3)

a) **How any iterations are needed to observe a reasonable convergence to a (fluctuating) minimum chain energy at this temperature?**

It turns out that this fluctuates significantly depending on which steps are taken (i.e. it depends on the random number generator). In certain cases the simulation tends to take the right steps and converge to a (local) minimum quickly, whereas in other cases it is a more gradual process. The reached minimum is also slightly different each time, which indicates that not all simulations reach the absolute minimum energy level. I expected this, since in some configurations a protein cannot be altered easily in such a way that it reduces the energy. Therefore they need large changes in order to get to an eventually lower energy level. In many cases such changes will first require changes increasing energy which are very unlikely to be accepted, hence such simulations become stuck at a local minimum.

For the given protein, energy levels between -12 and -16 are observed, with most of them -14 and -15 (at a temperature of 0.25). This indicates that in certain configurations, the system either somewhat stabilizes for a while until it finds a way to lower the energy level, or it becomes pretty much stuck on a fluctuating local minimum. This is not a stable minimum, since due to the fixed temperature, the probability of selecting a bad move is non-zero. However, in pretty much all of the cases the algorithm converges before 30.000 iterations, with most converging to a fluctuating (local) minimum before 15 to 20 thousand iterations. A very lucky example can be seen in figure 18c, where all of the H monomers have at least 1 neighbouring H monomer, which also converged relatively quickly and reached the suspected global minimum of -16.

b) **Does the minimum energy configuration correspond to a unique native state? In other words, if you repeat the simulation with the same parameters, will you obtain the same 'folded' conformations?**

No. As can be seen from the examples in figures 18 and 19, the energy level fluctuates and the global minimum is not always found. The final configurations are also very different, but they do expose some interesting patterns. Specifically, often similar groups of H monomers are folded 'against' similar groups of H monomers, often but not always in different configurations. This can be seen in figures 18a and 18c, specifically the similarity in their 'tails'. Some of these simulations get stuck in a local minimum, they have groups of folded H monomers in such a configuration that it's hard to optimize further, often with P monomers in between them. A clear example from this is 19c. It's clear that such simulations become stuck in local minimum which is hard to escape from. It's also apparent that simulations which converge in lower energy states, generally expose this behaviour less with regard to the same groups. Rather, those simulations get lucky and fold the whole chain in such a way that a lower energy state is found. They exhibit less separate grouping behaviour but get lucky and find a larger optimal grouping of H monomers, usually on the 'inside' of the protein. Figure 19c is a very good example of this, also being the lowest energy state observed of all simulations I have ran on this particular initial configuration. It can clearly be seen from figures 18c and 19a that simulations with many H monomers in the inside of the protein and P monomers on the outside is a good tactic for reducing the energy level.

c) **(How) does this change with temperature or the hydrophobicity of your protein?**

Reducing the hydrophobicity to 0.2 clearly changes the final results. Similar neighbour patterns between H monomers, see figures 20a middle top and 20c top left, seems to be more likely to occur. This makes sense, since there are much less possible combinations of neighbouring H monomers reducing the energy level. At the same time, while these simulations seem to quickly orbit around a local minimum energy configuration, from repeated testing they rarely ever reached their absolute local minimum energy configuration, never converging at it. They instead showed very erratic behaviour, a result of both a too high temperature and the low likelihood of energy-lowering moves making 'bad moves' more likely to be accepted.

Increasing the hydrophobic fraction clearly shows that some form of convergence almost always occurs within the first initial iterations. In most cases I observed a (local) minimum is reached with an occasional spike. An interesting property of the resulting configurations is that pretty much always P monomers were found at corners or endpoints, and only sometimes in the middle between H monomers. Rarely did I observe simulations with a more gradual decline to the minimum, see figure 21d. This makes sense, since there are so many energy reducing combinations to be found when the simulation starts, making it behave greedy at the beginning. While finding the easy steps favoring H monomers on the inside quickly, it also means finding the global minimum becomes harder. This can clearly be seen in the difference between figures 21a and 21b.

3.2 Simulated annealing

3.2.1 A note regarding the assignment instructions

Unfortunately this part of the assignment caused me a lot of confusion. Specifically, the phrasing in the assignment is really unclear and it's hard to understand how and what is exactly asked from us. Specifically the explanation on the SA algorithm is troublesome. I would advise to really improve that specific part of the assignment instruction for next year's course.

3.2.2 Initial configuration

For simulated annealing the same initial configuration as during the benchmark procedure has been used (See figures 15 and 16). This means a length of 25 monomers and a hydrophobicity of 50%. The initial temperature and amount of steps depends on the question being answered.

3.2.3 SA Algorithm and implementation details

The implementation of the Simulated Annealing algorithm is relatively simple. Effectively it takes discrete steps through a range of temperatures from some given maximum temperature T_{max} to a minimum temperature T_{min}. For every decrease in temperature it executes the *mmc* function on the given protein. This means that the temperature is decreased gradually and the probability of accepting a bad move becomes lower and lower. Similar to before the *mmc* function records the energy level every so often iterations. This part of the *mmc* function is modified so that it also records the radius of gyration, according to equation 1.

$$R_g = \sqrt{\frac{1}{N}\langle\sum_{k=1}^{N}(\vec{r_k} - \vec{r_c})^2\rangle} \tag{1}$$

N is the length of the protein chain. r_k is the position of the monomer at index k. r_c is the position vector indicating the centre of the monomer. The angular brackets indicate taking the average.
Effectively the average squared distance from the center of the monomer to each monomer is taken, which is then normalized and squared. The gyration radius is therefore a single scalar value representing the average normalized distance from the center, and gives an idea about the size of a conformation. The result of this computation is then also sampled every f steps, identical to sampling the energy, where f is the sampling frequency.

After the *mmc* function returns, the heat capacity is calculated from the results of the energy values. The first 10% of the resulting values are discarded in order to let the protein relax to the given temperature, before taking averages. The heat capacity is calculated according to equation 2.

$$C = \frac{\langle E^2\rangle - \langle E\rangle^2}{k_B T^2} \tag{2}$$

The angular brackets again indicate taking the average, in this case the average energy E. k_B is the boltzmann weight, in our case set to 1.0. T is the temperature at which the *mmc* function is executed.

3.2.4 Generated protein conformations

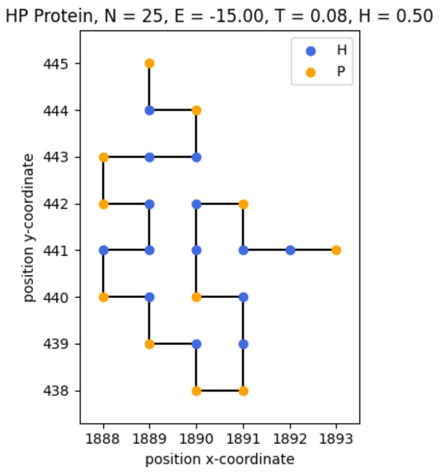
For this part the temperature was decreased to 0 starting at 2.0. This was done in 25 steps with 15000 MMC iterations per step. Due to the many figures only figures taken from two particular SA processes are presented here. However, multiple independent runs with different seeds were taken in order to verify correctness of these figures. Unless otherwise mentioned, figures on the left side will be from protein 1 and those on the right side from protein 2. All the graphs are generated such that the temperature is zero on the left side of the graph. This means that when it's mentioned that temperature decreases, the graph must be read right to left.

```
Protein 1

Final energy: -15.0
Lowest energy state found: -15.00
Mean energy state: -9.78
Lowest gyration radius found: 0.41
Mean gyration radius: 0.64
Resulting protein:
[((1893,441) P), ((1892,441) H), ((1891,441) H),
((1891,442) P), ((1890,442) H), ((1890,441) H),
((1890,440) P), ((1891,440) H), ((1891,439) H),
((1891,438) P), ((1890,438) P), ((1890,439) H),
((1889,439) P), ((1889,440) H), ((1888,440) P),
((1888,441) H),((1889,441) H), ((1889,442) H),
((1888,442) P), ((1888,443) P), ((1889,443) H),
((1890,443) H), ((1890,444) P), ((1889,444) H),
((1889,445) P)]
```

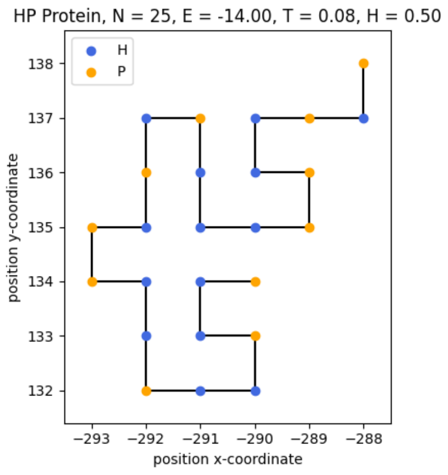
(a)

```
Protein 2

Final energy: -14.0
Lowest energy state found: -16.00
Mean energy state: -9.53
Lowest gyration radius found: 0.40
Mean gyration radius: 0.65
Resulting protein:
[((-290,134) P), ((-291,134) H), ((-291,133) H),
((-290,133) P), ((-290,132) H), ((-291,132) H),
((-292,132) P), ((-292,133) H), ((-292,134) H),
((-293,134) P), ((-293,135) P), ((-292,135) H),
((-292,136) P), ((-292,137) H), ((-291,137) P),
((-291,136) H), ((-291,135) H), ((-290,135) H),
((-289,135) P), ((-289,136) P), ((-290,136) H),
((-290,137) H), ((-289,137) P), ((-288,137) H),
((-288,138) P)]
```

(b)



(c)

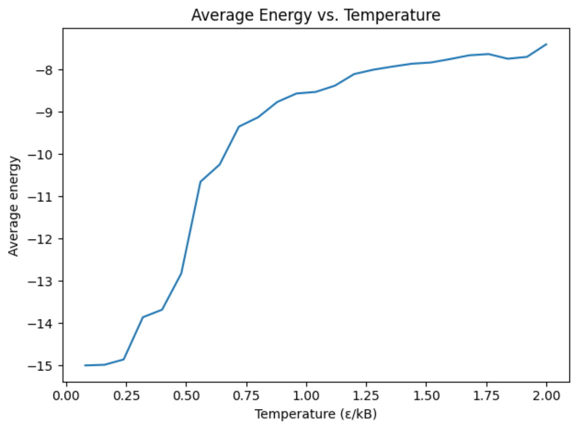


(d)

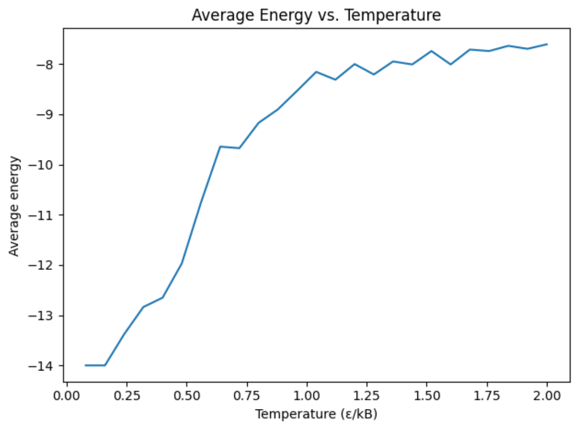
Figure 22: Resulting protein conformations 1 and 2

3.3 Energy and gyration radius distributions

The following figures are used for answering the questions related to the energy distributions.

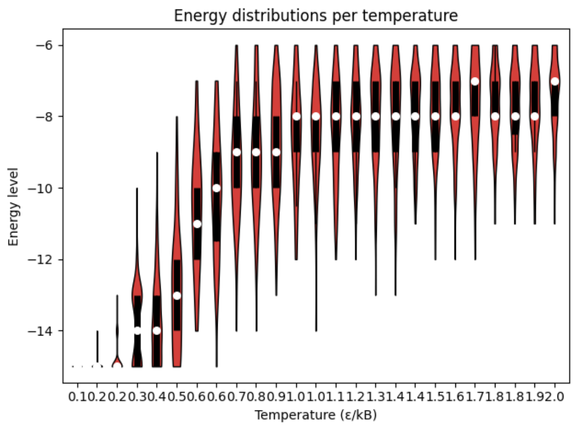


(a)

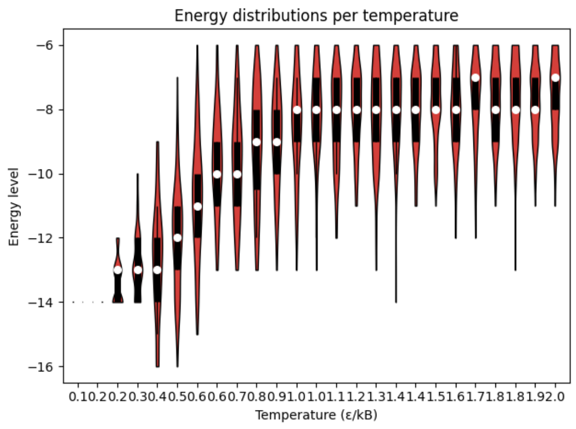


(b)

Figure 23: Average energy vs temperature of protein 1 and 2.



(a)



(b)

Figure 24: Energy distributions vs temperature of protein 1 and 2.

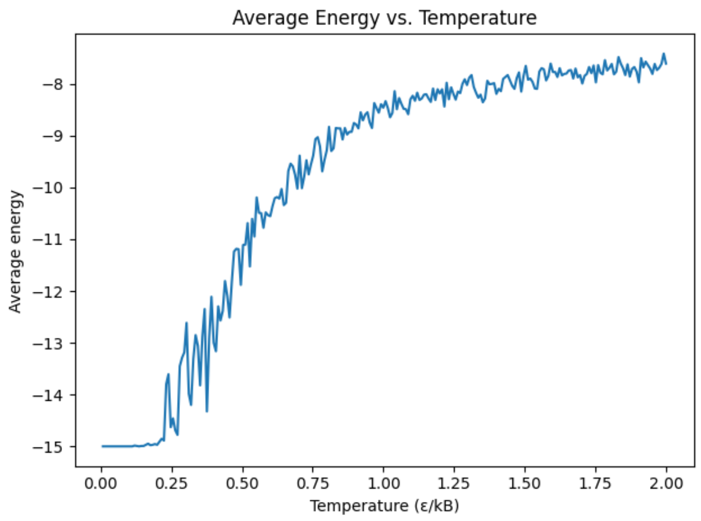


Figure 25: High resolution average energy vs. temperature curve of a protein.

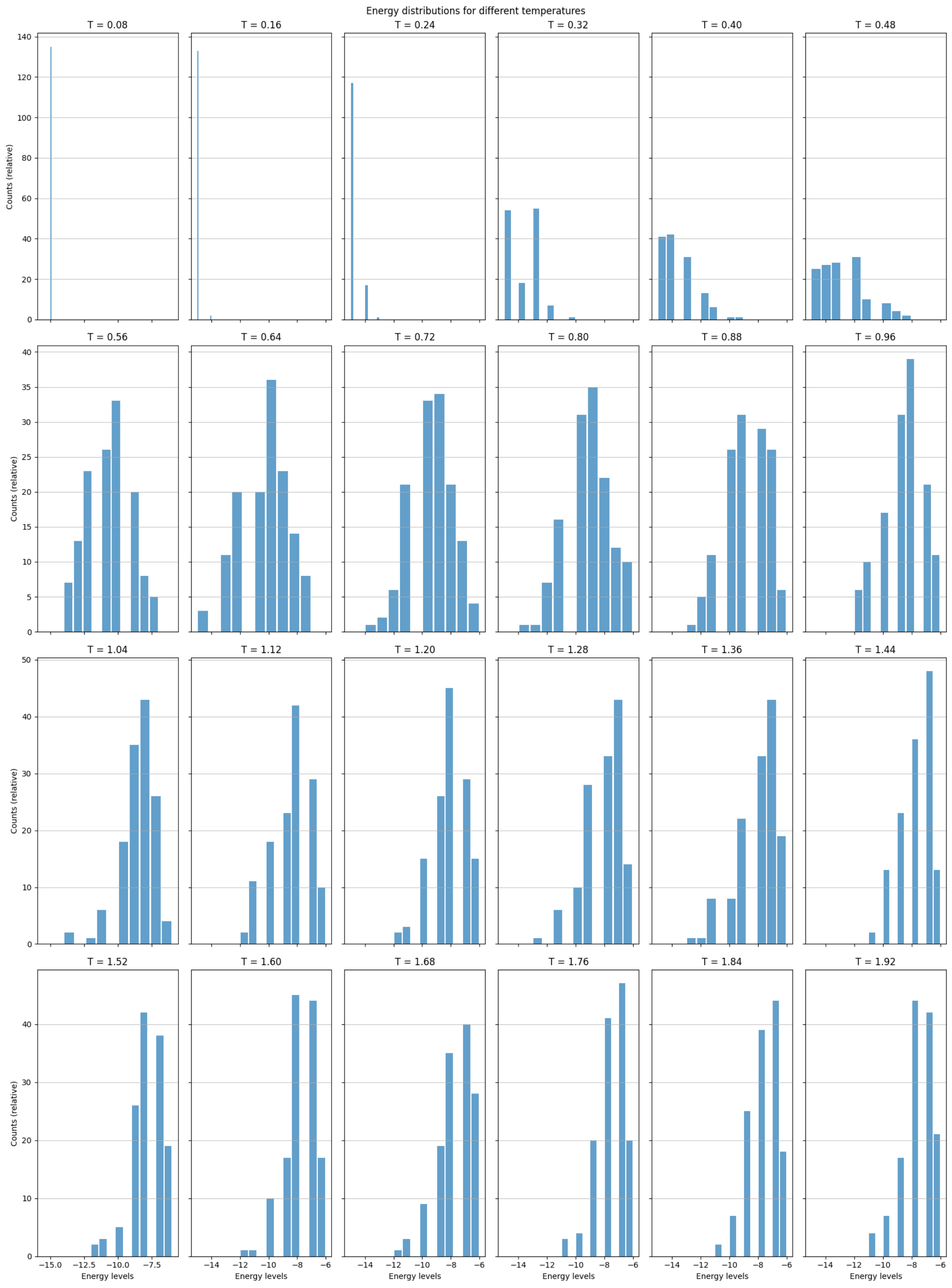


Figure 26: Distributions of energy over time for protein 1

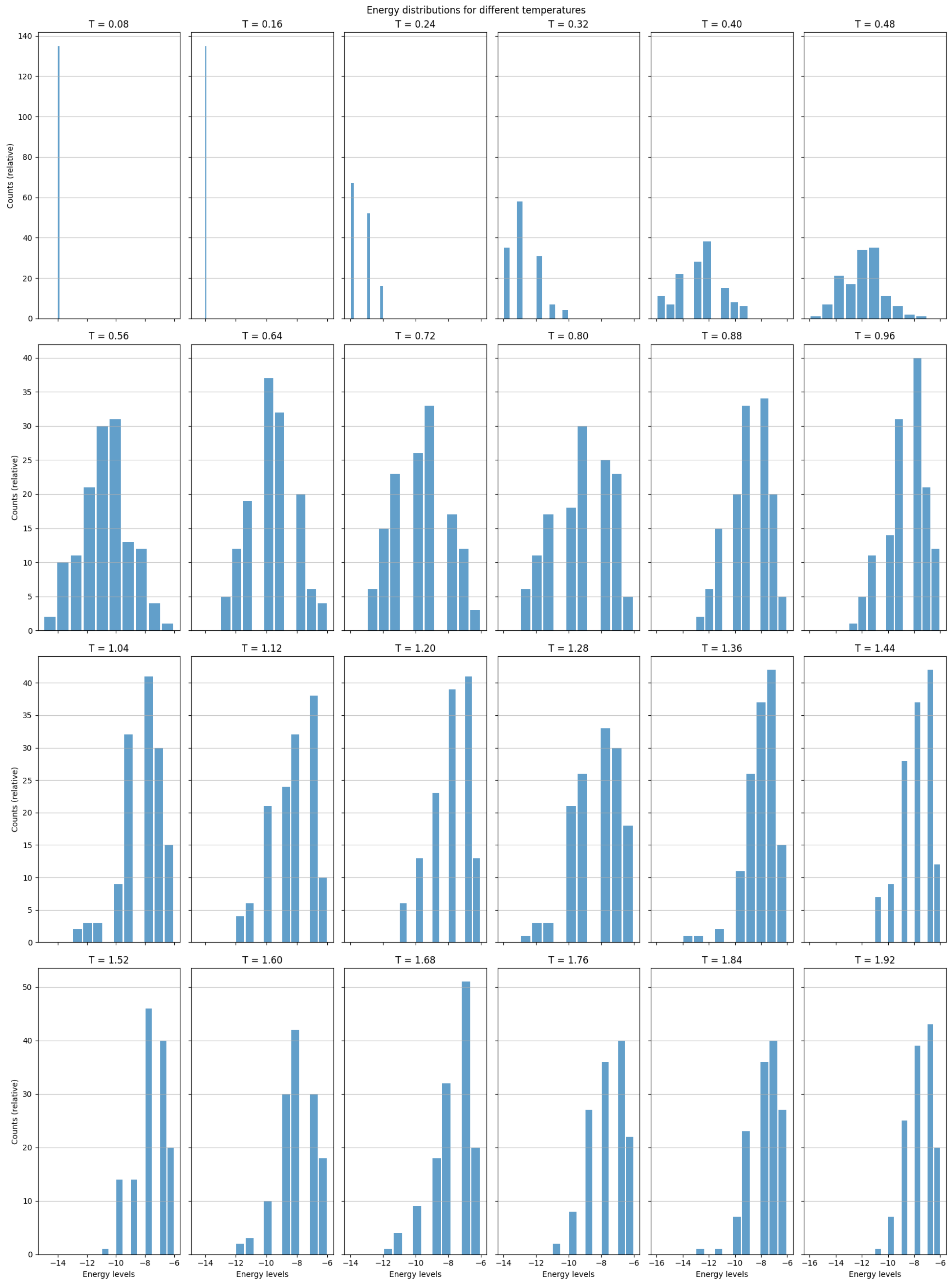
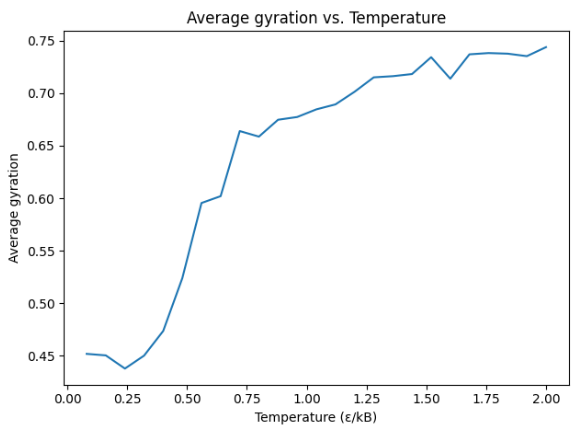
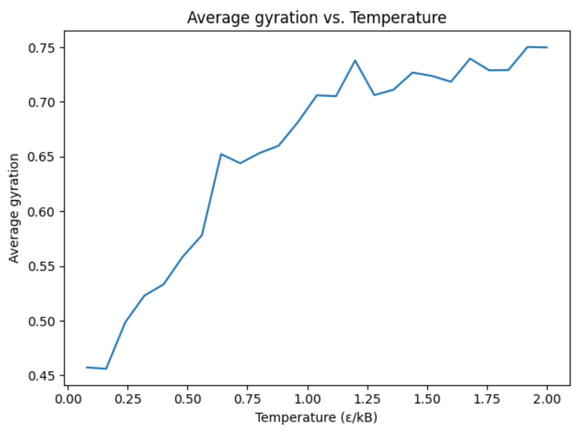


Figure 27: Distributions of energy over time for protein 2

The following figures are used for answering the questions related to the gyration radius distributions.

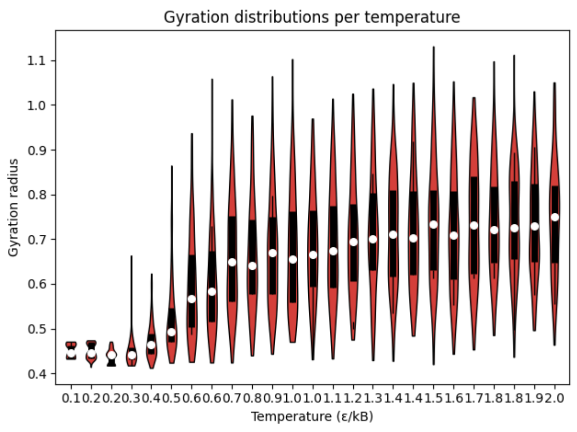


(a)

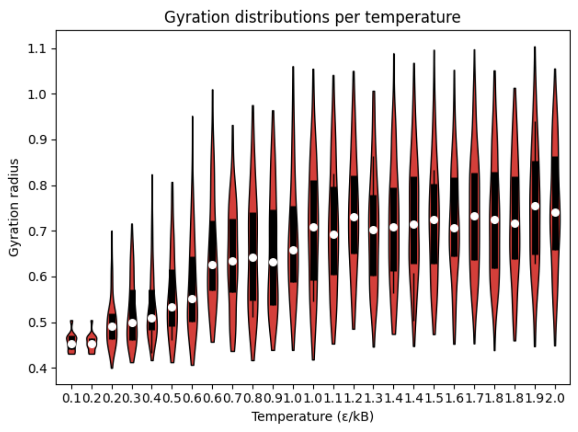


(b)

Figure 28: Average gyration radius vs temperature of protein 1 and 2.



(a)



(b)

Figure 29: Gyration radius distributions vs temperature of protein 1 and 2.

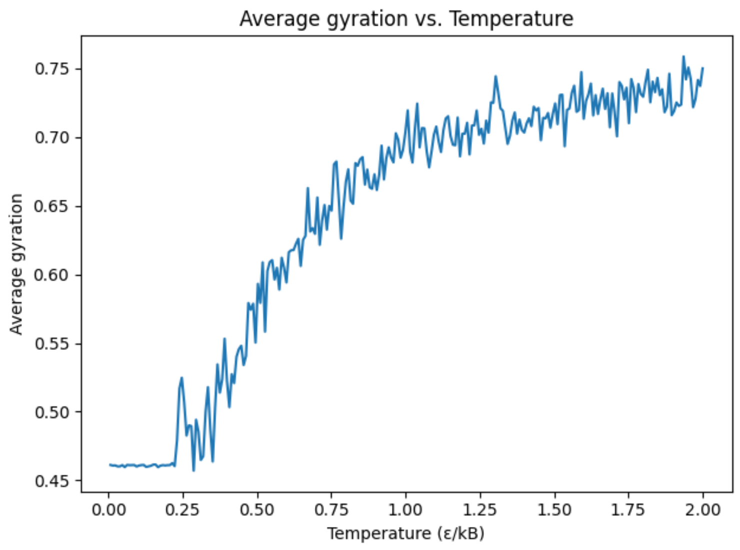


Figure 30: High resolution average gyration radius vs. temperature curve of a protein.

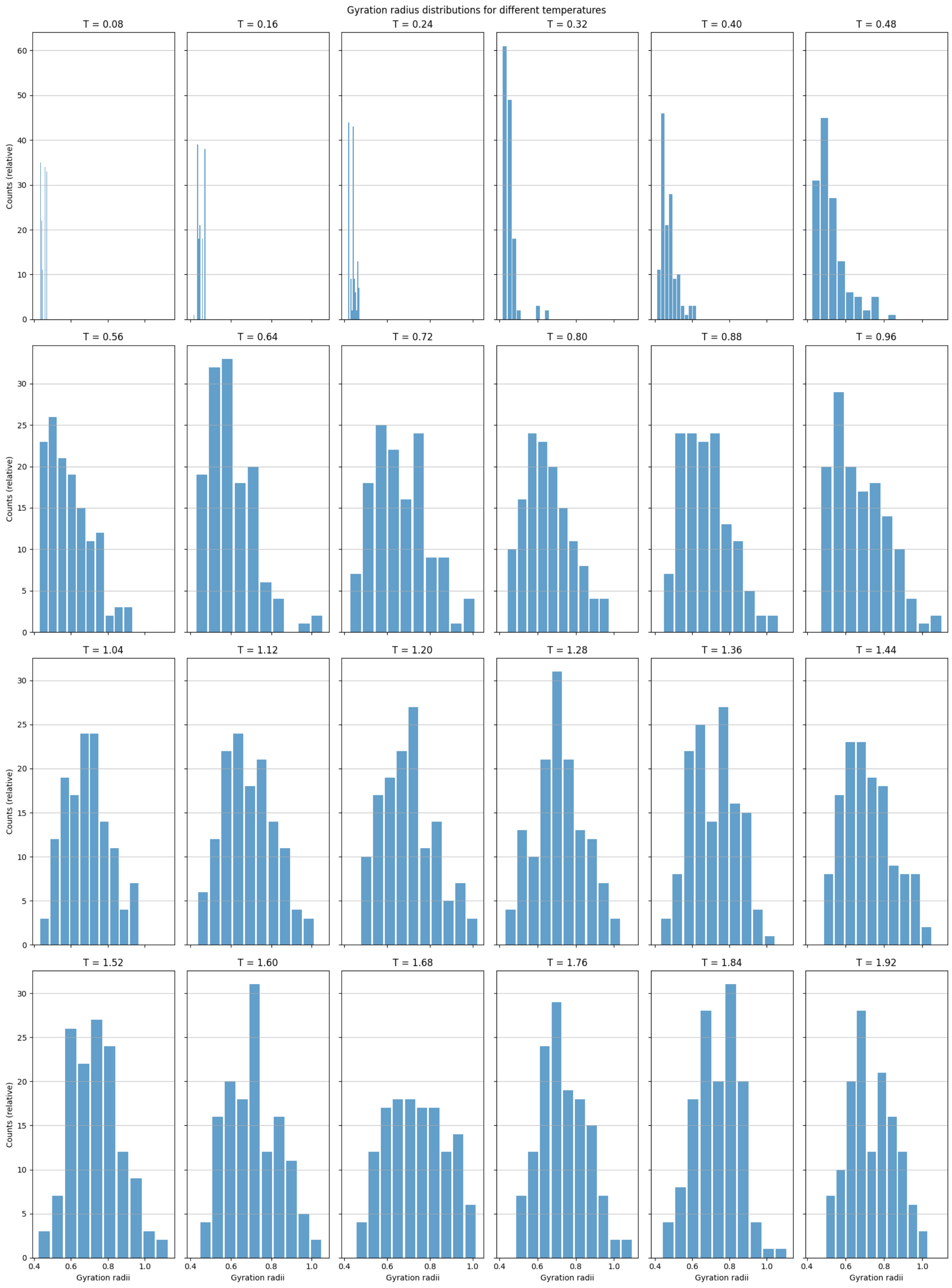


Figure 31: Distributions of gyration radius over time for protein 1

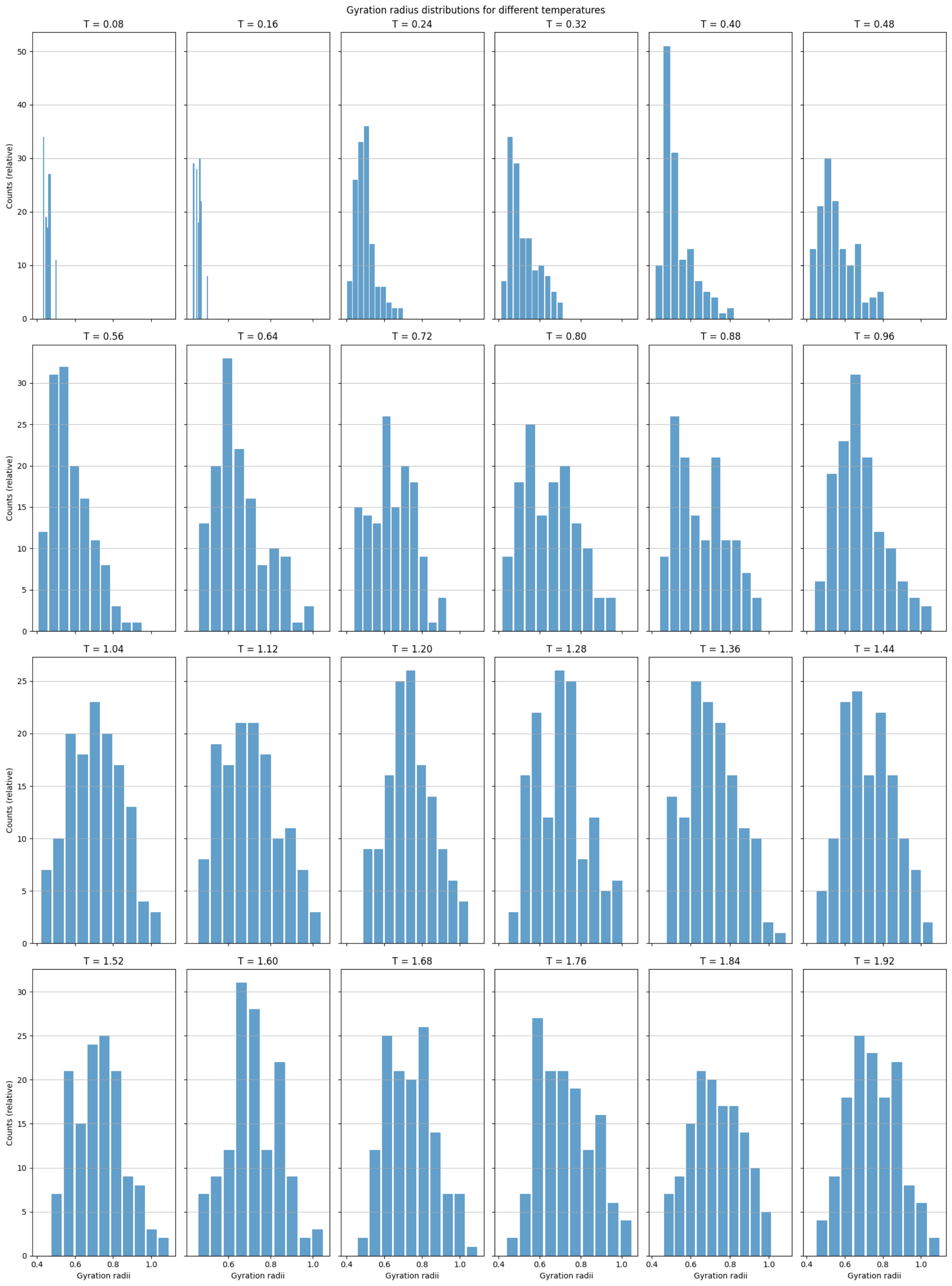


Figure 32: Distributions of gyration radius over time for protein 2

a) **How does the average energy distribution change over temperature? How does the width of the distribution change?**

It seems that the average energy level goes down as temperature goes down. This makes sense, since the probability of bad moves becomes smaller, thus slowly favoring better moves, and thus reducing the energy. It's clear from figure 23 that the energy level initially doesn't decrease very much, but as the energy decreases the slope becomes steeper, with a maximum steepness around $T = 0.5$, and becomes less steep again at very low temperatures. This makes intuitive sense since the probability of both accepting 'bad moves' and finding better configurations at very low energy levels become zero. Hence the variability in the different energy levels decreases and the slope stabilizes around $T = 0.15$.

Now let's look at the histograms generated for the distribution of energy levels over time. It's clear that the distributions themselves change shape significantly, as can be seen in figures 26 and 27. At high temperatures ($T > 1.5$) the probability of accepting bad moves is so high that the algorithm basically accepts changes randomly. However, it's clear that there is an upperbound to the energy level where the protein cannot reach higher levels. This causes the distributions to be skewed to the right when temperatures are really high.

As temperature goes down, the distributions move to the right, indicating that lower energy levels are explored. The width of the distribution also increases, since more varying levels of energy are explored. As temperature goes down, an interesting change happens around $T = 0.6$. At this temperature the distributions have moved to the middle of the graph and look like guassian distributions. They have also becomes less wide, indicating a shallower range of energy levels is explored.

As we decrease temperature even more, the curve starts to move to the right, favoring moves with lower energy levels more and more, until around $T = 0.2$ where the probability of accepting worse moves becomes so low that the protein practically doesn't change anymore. This is very visible in the high-resolution plot in figure 25. This indicates we have reached a minimum. From repeated testing it seems that this is not always the global minimum. However, it seems that it visits the global minimum at $E = -16$ more often than the *mmc* algorithm at a fixed temperature did, even if it doesn't stay at that temperature. Since in practical applications finding the global minimum is more important than finding it as final result, it's not much of a big deal that it doesn't always finish at the global minimum.

b) **How does the average gyration distribution change over temperature? How does the width of the distribution change?**

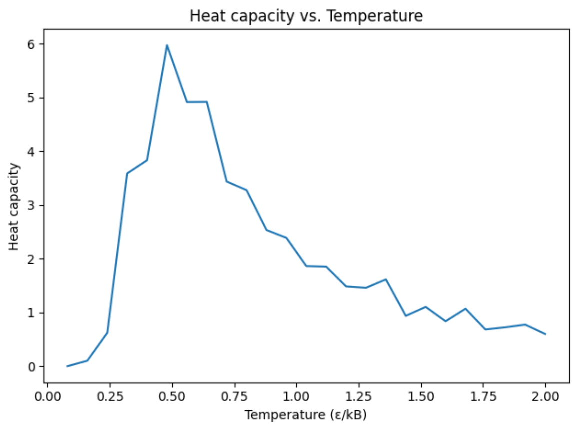
It is immediately clear from figures 28a and b that the radius of gyration also goes down as temperature goes down, but not quite in the same fashion as the temperature did. The change over temperature in radius of gyration is much more gradual than the change in energy was. Although at the end it seems to slightly drop faster until it slows significantly again around $T = 0.2$. The absolute change in gyration radius seems to also be relatively small as compared to energy levels. This clearly indicates that the protein slowly tends to fold 'inwards', reducing it's size, and thus increasing the likelihood of neighbouring H monomers lowering the energy.

At very high temperatures the distributions of gyration radius seem to be fairly stable (See figures 31 and 32). Due to the high probability of random changes being accepted, the radius of gyration varies quite a lot. At temperatures above $T = 1.5$ this seems to cause the distributions to be very gaussian like with very similar widths. This can also be easily seen in the relatively constant size of the violinplots in figures 29a and b. This makes intuitive sense, since for example very 'unfolded' conformations with maximum length are relatively unlikely to occur purely by random mutation. The same thing with very 'folded', and thus smaller, conformations. However, as temperature decreases, around the $T = 0.8$ the distribution of different radii seem to skew more to the left.

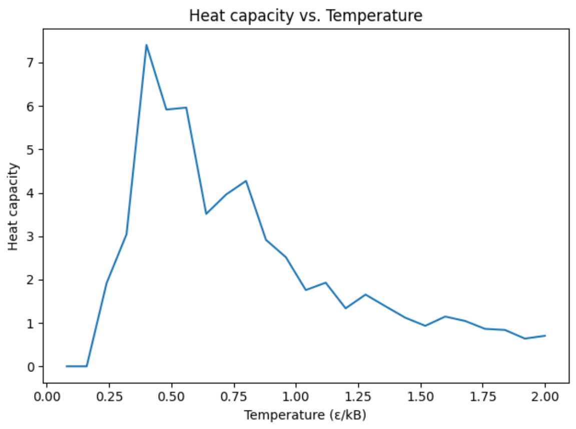
Around $T = 0.5$ the distributions start to skew very much to the left, counting many more occurrences of smaller radii. This causes the distributions to also become much thinner. This continues until around $T = 0.25$ where the distribution basically only contains a small range of radii. This is clearly visible in figure 30 as well. Obviously, there is a minimum size that the monomers can fit into, and it seems that this is around 0.4 (Keep in mind this is normalized). Similar to energy, there is a clear lower and upper limit to the energy levels and sizes the protein can take on. The slight reduction in gyration radius as energy goes down makes not only intuitive sense, but it also supports the phenomena I saw during the benchmarking part of the assignment, where I noticed that very low energy conformations tend to favor a group of H monomers in the centre. It also makes intuitive sense since the energy level is minimized when neighbouring monomers are neighbouring each other, and the amount of neighbours is highest when they are as clustered together as possible. There is a very obvious correlation between energy level and gyration radius as temperature decreases.

3.4 Heat capacity vs. Temperature

The following plots are created based on protein 1 and 2.



(a)



(b)

Figure 33: Heat capacity vs temperature of protein 1 and 2.

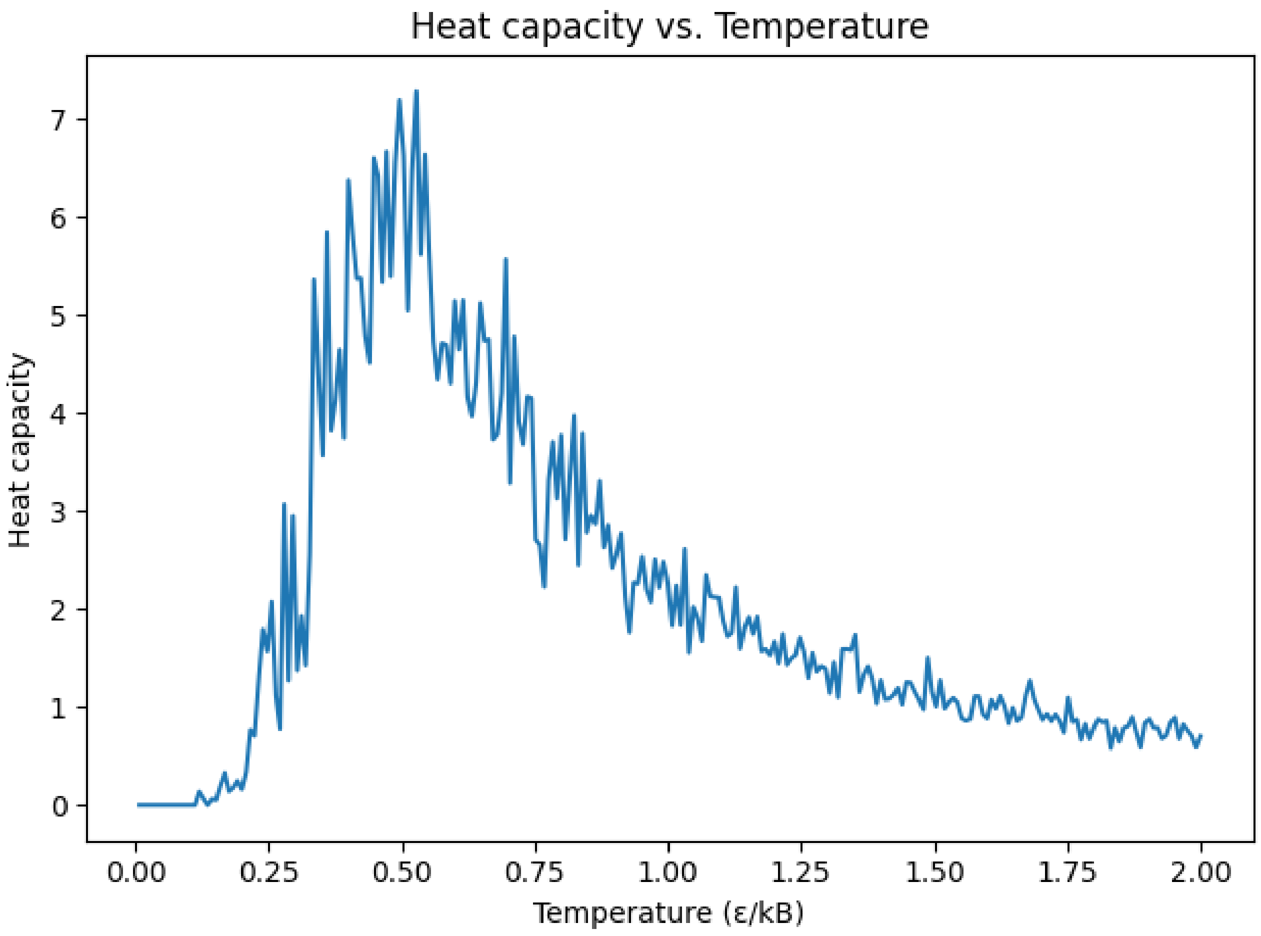


Figure 34: High resolution heat capacity vs. temperature curve of a protein using same seed as protein 2.

a) **At which temperature do you observe the peak in the heat capacity?**

From figures 33 and 34 it is very clear that the heat capacity peaks at around $T = 0.5$ for our given proteins with $N = 25$ and $H_{fraction} = 50\%$. They very clearly show a peak that indicates a heat capacity of somewhere between 6 to 7. Individual runs fluctuated slightly between 6 and 7.

b) **(How) can you relate the heat capacity plot to the average energy plot? And to the radius of gyration plot?**

Specifically in the plots regarding the average energy over temperature we already observed clear shifts to the left in the distributions of certain lower energy levels around $T = 0.5$. Specifically for energy it seems to be the point where the decrease in temperature corresponds to fastest decrease in energy, and thus the amount of different energy levels in the distribution quickly dwindles. This implies that around $T = 0.5$ the slope of the curve for average energy vs. time is the steepest.

The same thing effectively happens as well in the radius of gyration vs. temperature curves in figures 28 and 30. But again, this is sensible since lower energies usually correspond with a smaller radius of gyration. As can be observed in the histograms, the distributions of different radii of gyration also starts to skew a lot around $T = 0.5$. However, from repeated testing it seems this is less deterministic and more variable than the slope for the average energy curve. For example, from both the histograms (figure 32) and the curve (figure 28 it can be seen that for protein 2 the steepest points are slightly before and after $T = 0.5$. This is somewhat sensible since radius of gyration does not necessarily decrease if a move has been found which lowers energy. It might even occur that the radius increases. Although the general trend seems to be that the radius decreases as temperature and energy decrease.

c) **What does the peak in heat capacity indicate? Relate your answer to the plots you have produced.**

As mentioned previously, it seems that the peak in heat capacity approximately tells us where the steepest change in energy level occurs. The curvature we see also makes sense from a statistical perspective. At the lowest temperature, the entropy is zero, and thus there is very little probability that the protein can transition into a higher energy state. This is very noticeable from $T = 0$ to $T = 0.25$ in most of the figures presented before, but especially in 25.

The entropy goes up as temperatures increases, meaning that the probability of the protein to transition into a higher energy state goes up as well. At a certain point the temperature will approach the difference in energy levels, and a peak occurs. This occurs because at that point a small increase in temperature corresponds to a large change in entropy, and thus many more higher energy levels. What this means for us, in terms of the SA algorithm, is that once we reduce the temperature past this point, the search for more lower levels starts to narrow significantly. A good example of this is figure 23a, where a sharp slope can be seen at $T = 0.5$, indicating a massive increase in the various energy levels in the distribution at $T = 0.5$.

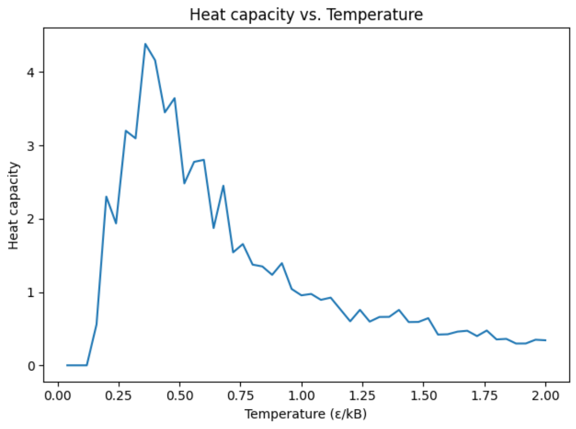
As temperatures increases even further, entropy takes over and the probability of each energy level occurring stabilizes. Therefore, therefore small changes in temperature again don't change really much anymore, and the heat capacity lowers and stabilizes. This can be clearly seen in 25 as the graph slows down significantly past $T = 1.0$.

After some research on the internet this effect seems to be called the hydrophobic force [3]. The paper also mentions that proteins with many H monomers tend to cluster into a group on the 'inside' of the protein when folded. An effect I have observed as well in my simulations. According to the paper this force has a peak around 60 °C to 80 °C. According to another source[4] this translates into a reduced units temperature of 0.5, which is exactly what I'm seeing in my simulations as well, confirming that this is indeed the effect I'm seeing.

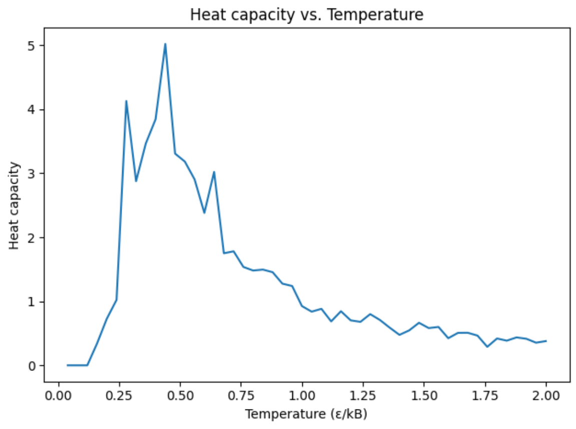
[3]https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4441443/
[4]http://www.acmm.nl/molsim/molsim2018/slides/n08-2_Lattice-Proteins.pdf

3.5 The role of hydrophobicity

a) Relation between hydrophobic fraction and peak temperature

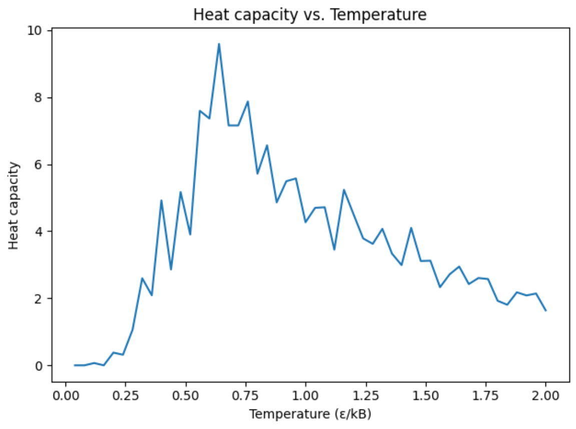


(a)

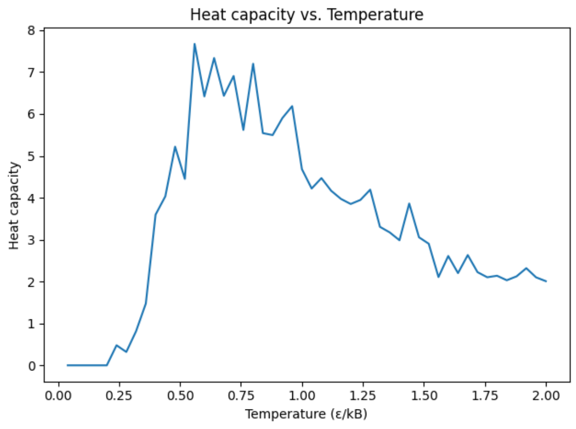


(b)

Figure 35: Heat capacity vs. temperature curves for H = 0.2



(a)



(b)

Figure 36: Heat capacity vs. temperature curves for H = 0.8

The above graphs are taken based on the initial proteins as used for the respective hydrophobic fractions in the benchmarking part of the assignment.

From the graphs it's clear that the absolute peak in heat capacity has a correlation with the hydrophobic fraction. The more H-monomers there are, the higher the peak in absolute heat capacity. Considering that the absolute energy is likely to be lower for proteins with more H-monomers due to more H-H contacts, and since heat capacity is directly related to the energy level per definition of it's formula 2, it makes sense that peaks are higher for higher hydrophobic fractions. The shapes of the curves don't seem to really change much, which is expected and is fine since there always will be some noise in them due to the involved randomness. However, the position of the peak seems to be ever so slightly altered as the hydrophobic fraction increases or decreases, but this could just as well be noise as a result of the randomness involved in the process.

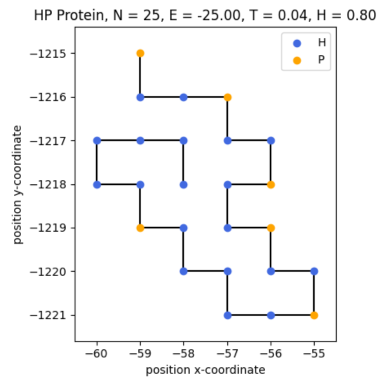
For the next part the following graphs and conformations are used, as well as earlier shown graphs and conformations for H = 0.5.

```
Protein 1

Final energy: -25.0
Lowest energy state found: -26.00
Mean energy state: -19.93
Lowest gyration radius found: 0.40
Mean gyration radius: 0.59

Resulting protein:
[((-59,-1215) P), ((-59,-1216) H), ((-58,-1216) H),
((-57,-1216) P), ((-57,-1217) H), ((-56,-1217) H),
((-56,-1218) P), ((-57,-1218) H), ((-57,-1219) H),
((-56,-1219) P), ((-56,-1220) H), ((-55,-1220) H),
((-55,-1221) P), ((-56,-1221) H), ((-57,-1221) H),
((-57,-1220) H), ((-58,-1220) H), ((-58,-1219) H),
((-59,-1219) P), ((-59,-1218) H), ((-60,-1218) H),
((-60,-1217) H), ((-59,-1217) H), ((-58,-1217) H),
((-58,-1218) H)]

Lowest protein:
[((-773,-815) P), ((-773,-814) H), ((-773,-813) H),
((-772,-813) P), ((-772,-812) H), ((-773,-812) H),
((-773,-811) P), ((-774,-811) H), ((-775,-811) H),
((-775,-810) P), ((-776,-810) H), ((-777,-810) H),
((-778,-810) P), ((-778,-811) H), ((-777,-811) H),
((-776,-811) H), ((-776,-812) H), ((-776,-813) H),
((-776,-814) P), ((-775,-814) H), ((-775,-813) H),
((-775,-812) H), ((-774,-812) H), ((-774,-813) H),
((-774,-814) H)]
```

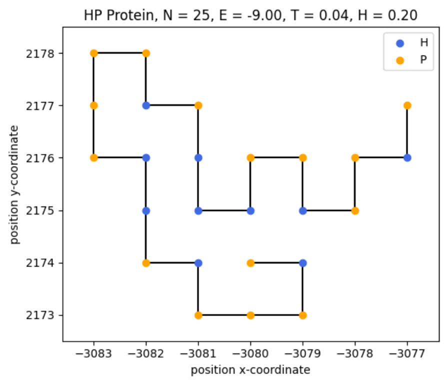
(a)

```
Protein 2

Final energy: -9.0
Lowest energy state found: -9.00
Mean energy state: -5.11
Lowest gyration radius found: 0.41
Mean gyration radius: 0.68

Resulting protein:
[((-3080,2174) P), ((-3079,2174) H), ((-3079,2173) P),
((-3080,2173) P), ((-3081,2173) P), ((-3081,2174) H),
((-3082,2174) P), ((-3082,2175) H), ((-3082,2176) H),
((-3083,2176) P), ((-3083,2177) P), ((-3083,2178) P),
((-3082,2178) P), ((-3082,2177) H), ((-3081,2177) P),
((-3081,2176) H), ((-3081,2175) H), ((-3080,2175) H),
((-3080,2176) P), ((-3079,2176) P), ((-3079,2175) H),
((-3078,2175) P), ((-3078,2176) P), ((-3077,2176) H),
((-3077,2177) P)]

Lowest protein:
[((-2191,651) P), ((-2191,652) H), ((-2190,652) P),
((-2190,653) P), ((-2189,653) P), ((-2188,653) H),
((-2188,652) P), ((-2187,652) H), ((-2187,653) H),
((-2187,654) P), ((-2187,655) P), ((-2186,655) P),
((-2186,654) P), ((-2186,653) H), ((-2185,653) P),
((-2185,652) H), ((-2186,652) H), ((-2186,651) H),
((-2186,650) P), ((-2187,650) P), ((-2187,651) H),
((-2188,651) P), ((-2189,651) P), ((-2189,650) H),
((-2190,650) P)]
```

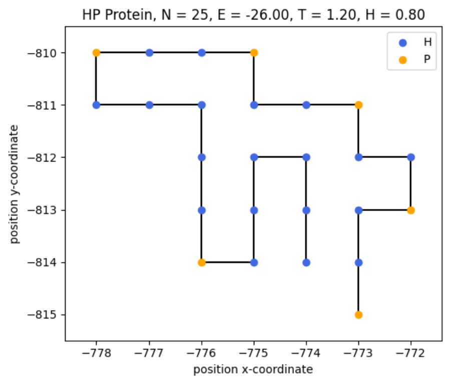
(b)



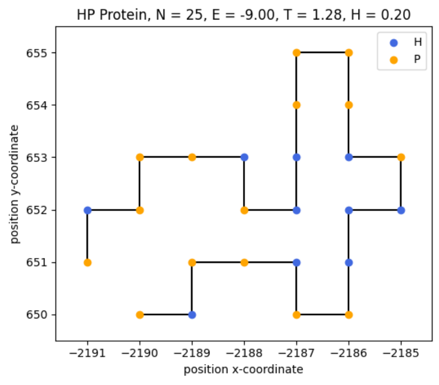
(c)



(d)

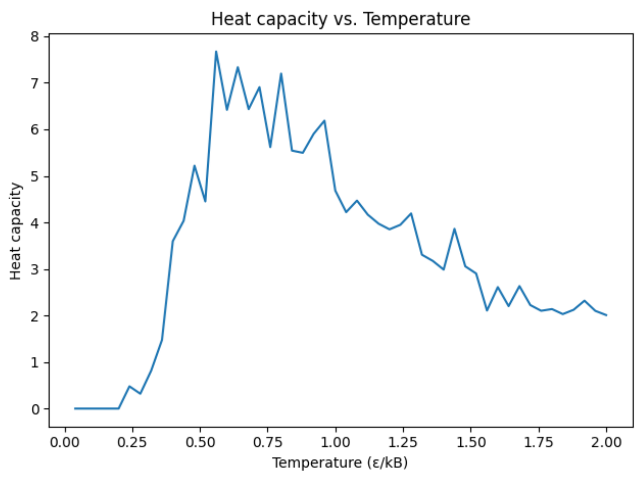


(e)

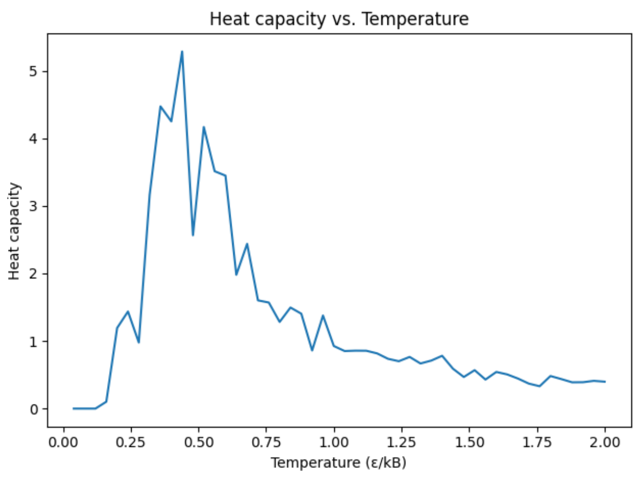


(f)

Figure 37: Resulting and Lowest protein conformations 1 (H = 0.8) and 2 (H = 0.2).



(a)



(b)

Figure 38: Heat vs. Temperature for protein conformations 1 (H = 0.8) and 2 (H = 0.2).

b) **Conclusion on the folding process based on hydrophobic fraction**

In order to generate the resulting graphs for the lowest found energy state, code was added which kept track of the lowest energy conformation that was found. For implementation details on this I would like to refer to the provided source code.

From the above graphs it's again very clearly visible that proteins tend to group H-monomers in the center. In all the conformation plots in figure 37 this is clearly visible. From the graphs in figures 38, 33 and 34 it's clearly visible that the peak of the spike is higher depending on the hydrophobic fraction. It can also be observed that the final conformations are not necessarily the most optimal ones with the lowest energy states. From the conformation plots it's also immediately clear why we can observe the gyration radius to shrink as temperature becomes lower. The proteins clearly show clustering behaviour, which is a result of attempts to lower the energy. This also nicely fits into why I suspect it's called a hydrophobic fraction. Suppose we suspend our protein in water. The hydrophobic monomers (H type) don't like water, so they move to the inside of the protein, becoming shielded from the water by the P monomer types.

4 Conclusion

The goal of the assignment is to implement an algorithm which simulates protein folding in order to find the lowest energy state of a HP protein on a 2D lattice. This has been done using the Metropolis Monte Carlo algorithm which randomly perturbs the protein chain accepting changes which decrease the energy level. Acceptance of changes increasing temperature is controlled through a temperature parameter. Simulated Annealing is the process in which this temperature parameter is gradually reduced in order to more efficiently search through the solution space of moves.
It's clear from the various results that in general lower energy levels are reached when as many as possible H-monomers are in the 'inside' of the protein. As such, the energy level and gyration radius have a clear correlation with each other. The heat capacity peaks at the point where the probabilities of accepting better changes and worse changes converge. In other words, the peak in heat capacity occurs when a small change in temperature causes a massive change in the distributions of the various energy levels being visited. Finally, the hydrophobic fraction clearly affects the peak value of this peak in heat capacity. Higher hydrophobic fractions cause the peak value to become higher, lower fractions lower the peak.
In general the MMC and SA algorithms that were implemented tended to not always find the lowest energy state of the protein. MMC was, as expected, very sensitive to the temperature parameter. Simulated Annealing clearly improves this and found the global minimum far more often. However, it's also clear that a heuristic approach specifically built to optimize H monomers on the 'inside' might be a far more efficient approach in finding the global minimum, as this global minimum is directly related to the H monomers being clustered in the centre of the protein.

5 Possible future improvements

One of the ways to improve the accuracy of the results that was attempted was to create a multi-threaded simulator which created weighted averages for the various curves. Due to the relatively long runtime of a single SA run, I had to parallelize this to get results in a reasonable timeframe. The results were computed by taking a certain amount of samples of a few separate SA runs and using the averages to compute a more accurate estimate for the heat distribution curve. It turned out however that this wasn't really worth it, it definitely got better and more accurate results, but the amount of samples required combined with a reasonably large amount of temperature steps meant that computation time was still in the order of tens of minutes while running 12 samples simultaneously. Perhaps smarter weighting of samples could reduce the amount of samples, but in practice fitting a curve is likely a better approximation than a lot of wasting computation time.

Another specific part I would like to play around with in the future is the way the temperature is reduced. While I played around with gradual linear (no discrete steps) reduction of temperature, this didn't significantly affect results. However, I can imagine certain other distributions by which to lower the temperature might yield better results in some cases.

The final suggestion which could be interesting to explore in the future is to create an algorithm that specifically and intelligently optimizes and selects moves such that H-monomers fold to the 'inside' of the protein. Since clearly, the formation of H monomers on the inside is key in reaching lower energy states. A more intelligent algorithm exploiting this could therefore more efficiently find the global minimum and thus the native state of the protein.