# Protein folding with nearest neigbour contribution

Stdudent id: 10053

May 5, 2023

In the following article, the data structure I implemented for the exam will be explained, and the results of running the code will be discussed. The report contains my answers to every question in the prompt, with the exception of the last one, which I did not have time to implement. The Metropolis Monte Carlo method, in combination with Simulated Annealing, was used to find tertiary structures for different proteins, and the method used extends to 3D.

## 1 Introduction

Protein folding is a complex and crucial process that determines the three-dimensional structure of a protein, which in turn influences its function within living organisms. Understanding this process is vital for numerous applications, such as drug development, protein engineering, and understanding the molecular basis of diseases. In this report, I use my implementation of protein folding with a focus on nearest neighbor interactions to generate several results about Simulated annealing of proteins and their phases. The implementation will be elaborated in the methods section.

To facilitate clarity and ease of evaluation, I have organized the report in such a way that the answers to tasks 1 and 2 are clearly indicated at the beginning of each relevant subsection in the methods and results sections. This organization enables the reader to quickly locate and assess the completion of each task, ensuring a more comprehensive understanding of our findings and their implications.

## 2 Theory and Questions

In this section i have covered the answers to the preliminary questions, Which i felt fit as theory, as some things written are referenced later.

### 2.1 Possible tertiary structures

To answer the first question i will make an approximation. I will not consider if the protein loops back on itself later. Assume there are N monomers that should be put together in a chain. Start out with the first monomer. This monomer becomes the reference (origin), so it can only have one position. The next monomer can be placed in any of the four unit directions around the origin (positive and negative x and y direction). Now consider the vector $v_n$ pointing from the position of the n-1-th monomer to the position of the n th monomer. This vector will be of unit length and defines the forward direction of the next procedure. Now the n+1th monomer can now be placed to the left, right or forward from the position of the nth monomer. In reality some of these positions will be invalidated by the fact that one or more of them may be occupied already. Finally proteins rotated by $n\pi/2$ will be considered equivalent, so the result is divided by four. Therefore the number of unique proteins of length N, not accounting for possibly overlapping structures is $min(1, 3^{N-2})$. In 3D this number jumps up to $min(1, 6 \cdot 5^{N-2}/|C|)$, ($|C|$ is size of the rotational symmetry group of cube) Due to the fact that we can also go up and down.

### 2.2 Time to check every structure

Assuming there are 300 monomers that should be put in a chain, we find that the possible number of proteins is of the order $3^{298} \approx 1.995 \cdot 10^{140}$. To put this into perspective, current the lifetime of the universe is of the order $4.355968 \cdot 10^{17}$ seconds. So even if it took a computer $10^{-12}$ seconds to try each structure, it would still about 100 orders of magnitude longer than the current lifetime of the universe to consider all the possible states.

### 2.3 Ints and floats (doubles)

The answer to the question of the largest integers and floating-point numbers a computer can store depends heavily on the system the computer is running and the programming language used. In my case, the system uses standard ints and doubles (double-precision floating-point numbers used in C++) that occupy 4 and 8 bytes, respectively. This means that, for code in C++, where the compiler uses the system standard, it can store ints as large as $2^{32-1} - 1 = 2,147,483,647$ and doubles as large as $(2 - 2^{-52}) \cdot 2^{1023} \approx 1.8 \cdot 10^{308}$. In Python (version 3, not version 2), the integer type has arbitrary precision and is only limited by the available system memory [1]. It is therefore much harder to estimate the largest possible int in Python. However, the floating-point numbers (float) in Python use 8 bytes on my computer.

### 2.4 floats and doubles

When adding or multiplying floats and doubles, one can either say that one loses the precision of the double or that one preserves the precision of the float. When working with significant digits, if one adds a number with N significant digits to a number with 2N significant digits, the uncertainty of the first number in the Nth digit dominates the uncertainty of the second number.

When a float and double are added, the float is cast to a double, and the extra bits in the mantissa are filled with zeros. So, the result is a double, but the precision of this result is really only as large as the float's precision. I am here assuming the numbers are of a similar order.

A similar problem occurs for multiplication. In multiplication, the exponents are added, and the mantissas are multiplied by summing over bit-shifted versions of the first mantissa, depending on the bit in the second mantissa. Therefore, the uncertainty in the float affects the result.

In conclusion, if one considers floats and doubles in computers as ranges with uncertainty in all the digits that would come after the precision of the mantissa, one can see that adding or multiplying floats and doubles would kill the precision of the double.

### 2.5 Smallest n such that: $1.0 + n \neq 1.0$

For addition in floating-point arithmetic, the largest number is taken as the first number. The second number has its exponent adjusted to equal the first, and the mantissa is bit shifted by the difference of their exponent. Any of the least significant bits are lost, so the operation will be less precise the larger the difference between the two numbers. This is generally why backward sums are more precise than forward sums, as the small contributions in the backward sum are allowed to cascade instead of being chopped off. Now, if the number is so small that the bit shifting in the mantissa results in a zero mantissa, then the sum of the two numbers will equal the largest. The answer therefore depends on the length of the mantissa. For floats, this smallest n is approximately $1.192 \cdot 10^{-7}$, and for doubles, it is approximately $2.220 \cdot 10^{-16}$.

## 3 Method

### 3.1 Choice of programing language

For this project i have chosen to work with both python and C++. Python has the benefit of being pretty, easy and having a good library for plotting data (matplotlib) and a library for easy, and comparatively fast data handling library (numpy, which is fast due to its implementation i C++). C++ on the other hand is blazing fast due to it being a compiled language, which in retrospect was invaluable during the project. In some of the simulations i am doing of the order of a billion function calls (Simulated annealing and phase change), and my computer is still able to actually preform the simulations. However C++ is nontrivial to program in, sometimes requires multiple lines to write things Python does on one, and and does not have a equivalently good plotting library which i am familiar with.

My solution is to simply preform the complex calculations i C++ and writing the plotting and data handling functions in python. The two communicate with each other by the C++ methods writing their data to files, and having the program in python read in the data from the files. This solution gives me some of the benefits of both the languages, however it requires me to write multiple scipts for importing and exporting the data. The required functions are luckily quite simple, and as this has been my solution in multiple assignments i have become quite proficient in creating them. I am however hoping for the day that python gets a jit compiler which one does not have to babysit to get to work (numba).

### 3.2 ProteinSystem

**relevant for 2.1.1, 2.1.2, 2.1.3, 2.2.1** The class ProteinSystem is defined in the header ProteinSystem.h. It contains several variables which are important for the system, however the ones most relevant here are `N`, `xPositions`, `yPositions`, `zPositions`, `is3D`, `aminoAcidType`, `r`, `interactionMatrix` and `nearestNeigbours`.

The positions of the amino acids are stored in the position vectors as ints. This is because the distance between the monomers are fixed, and furthermore the angles between them are either $\pi/2$ or $\pi$. Therefore i could simplify the positions as integer vectors. This is not the only data structure i considered. Similar to in 2.1 i could have stored only the direction the next monomer relative to the last. This implementation would have some strengths my current solution does not have but also some additional drawbacks. I will come back to this alternative data structure later.

The system has two constructors, the second of which is relevant. It takes in number of particles `N`, an interaction matrix, which is implemented as a `vector<vector<int>>` and two booleans `Random` and `is3D`. `is3D` is used by the system to determine if it is in 2D or 3D mode. In the methods where this is relevant, `is3D` determines the behaviour of the system. `Random` determines if the system should be initialized as a straight protein in x direction or if the next monomer in the sequence should have have position 1 away from the previous monomer in any of the cardinal directions, assuming the position is not occupied.

To get random numbers, my system uses a random number generator from GSL (Gnu Scientific Library), seeded with the time when `ProteinSystem` is constructed. It has to be deallocated in destructor, as it is a pointer. RNG is first used to initialize `aminoAcidType` (which is a `vector<int>`) with numbers between 0 and 19 representing the different amino acids. The amino acid type can also be set manually by a setter function. if system is in random mode, it will then overwrite the old positional data. It finds possible positions of the next monomer in the chain with generatePossiblePositions, which returns a `vector<vector<int>>` containing allowed positions. This method takes into account if `is3D` is true or not, and generates proper in the way described in end of the above paragraph.

Finally the constructor calls the generateNearestNegbours method, which constructs the nearestNeigbours member variable. This is a `vector<vector<int>>` of dimensions (a,N), where a is 3 for 2D and 5 for 3D. It stores the indices of every nearest neighbour for each monomer. In 2D for most of the monomers the number of possible nearest neighbours is 2, as two of the possible nearest neighbours has covalent bonds with the monomers (and are not included as Proper nearest neighbours). In 3D this number is instead 4. The end points however, can possibly have an additional nearest neighbour. Therefore to have all the lists the same length each get a set of 5 or 3 indices depending on `is3D`. If a monomer of index i is neighbor with monomers with index j and k, then both j and k are stored in `nearestNeigbors[i]`. The rest if the elements are set to -1, which would not be a possible index. `nearestNeigbors[i]` $= (j, k, -1)$ in 2D mode.

The `generateNearestNeigbours` method runs over all indices, an for each index it calls `determineNearestNeigboursIndexI`, which takes in both the positions and index of monomer with index i and returns the indices of neighbouring monomers. This submethod is used, so that the code can be reused by Monte Carlo algorithm.

## 3.3 generateInteractionMatrix

**relevant for 2.1.2** My interaction matrices are generated by `generateInteractionMatrix`. It is uses GSL to fill a 20x20 matrix (`vector<vector<double>>`) with random numbers between -2 and -4. It uses GSL to generate the random numbers. We want the interaction matrix to be symmetric, which implies that it is equal to its transpose $M_{i,j} = M_{j,i} \quad \forall i, j$. It therefore runs over all i, and all j smaller than or equal to j. We want symmetric matrix, as the energy in the bond between two types of amino acid should not be different based on which is considered the first or second amino acid. The interaction is symmetric, so the matrix should be as well.

An interaction matrix can be written to a .txt file by using `writeInteractionMatrixToFile` or read form file using `readInteractionMatrixFromFile`. At the start of the project i generated one interaction matrix, which i wrote to a file. This was used for all subsequent simulations, with the exception of 2.2.9, where i copied the old file and manually wrote in some positive numbers into it.

## 3.4 nMonteCarloSweep

**relevant for 2.1.5, 2.1.6 and 2.2.2.**
`nMonteCarloSweep` takes in an integer `n` and preforms n Monte Carlo sweeps. A in one Monte Carlo sweep the system makes `N` (number of monomers) calls to the `monteCarloDraw` method. After each sweep the system calculates the energy of the system, the radius of rotation and the end to end distance, and stores the data in a data struct called `MonteCarloData`.

`MonteCarloData` has a method for exporting itself to a file, and a + operator, which creates a new inctance, and appends the second onto the end of the first. This is usefull both for `anealSystem` and for if one wants to use `nMonteCarloSweep` on a system twice, and merge the data.

The `monteCarloDraw` method selects at random one of the indecies and then calls `determineLegalMoves` which returns a `vector<vector<int>>` which contains the coordinates of each new move a monomer can make. Generaly this list has one element, however if the drawn index denotes one of the end points the vector could potentially contain 3 or 5 elements depending on if `is3D` is `true` or `false`. So it is written as such for flexibility. It could also be empty, if the monomer has no legal moves, in which case the method returns. If it has elements then a random new position is chosen. The function then calculates the change in energy between the system using `interactionMatrix` and the member variable `E` which is initialized by the constructor. Finally it performs the

Metropolis Monte Carlo algorithm. The algorithm always accepts the result if it reduces the energy of the system, and it accepts or rejects the new position probabilistically based on the temperature of the system and the difference in energy between the old and new state.

The legal moves are determined in `determineLegalMoves` by first checking whether or not the index is one of the end points. If it is not, then it calculates the position of the opposite corner from itself by the following formula: let $r_i$ be the positional vector of the ith monomer. Then the corner will be the sum of $r_i$, the pointing from monomer with index i to monomer with i-1, and the vector the pointing from monomer with index i to monomer with i+1. This result in vector $r_i^{new}$ pointing to the opposite corner defined by equation 1.

$$r_i^{new} = r_{i-1} + r_{i+1} - r_i \tag{1}$$

Then the method runs over all indices and checks if $r_i^{new}$ is occupied. If so then the method returns with an empty vector. Note that if monomer with index i is not a corner, then $r_i^{new} = r_i$, which of course is occupied by itself. Thereby the monomers in the middle only move if they are corners. I considered whether the middle monomers actually had an additional legal move. If the monomers where allowed crank moves, by which i mean that the monomers would be allowed to change the angle between the vector pointing from it to the previous monomer and the vector pointing from it to the next by $\pi/2$ radians, then the protein would probably fold much faster. Such a transform would in my implementation have required the use of rotation and translation matrices, which would be painfully slow. In the anther data structure i mentioned in section 3.2 this would have been trivial, however other calculations would have been harder. i decided not to implement it as moving the entire protein by $\pi/2$ radians instantly felt quite nonphysical, and it was not explicitly mentioned by the assignment.

The first and last element can move anywhere relative to the monomer it is connected to, provided the positional data are still ints and the positions are not occupied. In 2D mode this equates to 4 initial positions, which are reduced by at least one because of the monomer it is connected to. in 3D mode the monomer is also allowed to move one unit in the positive or negative z direction. Thereby yielding 6 initial positions which again is reduced by at least 1. Note that the `zPosition` array is still used by the 2D system, however as moving in the y direction is never allowed in 2D mode, resulting in `zPosition[i]` $= 0 \quad \forall i$

## 3.5 anealSystem

**Covers the method used in 2.1.7, 2.1.8, 2.1.9 and 2.2.3** `anealSystem` takes in the following two ints: `itersBeforeTempChange` and `tempsteps`. These integers can be used to get data on what phases the protein has, or it can be used for
simulated anealing.

The method starts the system out at 10 degrees. Then a vector of temperatures is created. This vector has length `tempsteps`, and it is implemented similarly to linspace in python, only in descending order towards zero, and the system never actually reaches exactly 0. Then the system runs over all temperatures in the temperature vector and for each temperature in `tempsteps` the system temperature is set to temperature. The method then calls `nMonteCarloSteps` with `itersBeforeTempchange` as its input. The `MonteCarloData` from `nMonteCarloSteps` is then averaged over the last half of the indecies. This is to let the system settle into its new temperature before continuing. This average is stored in a data structure called `PhaseCangeData`, which is similar to `MonteCarloData` only that the sweep numbers are not stored, but instead the temperatures are stored. This struct also have a method for writing the contained data to a file. `PhaseChangeData` is at the end returned by the method.

If one wants high resolution `PhaseChangeData`, `tempsteps` can be chosen as a small number whilst `itersBetweenTempSteps` is chosen to be very large. For simulated annealing one does the opposite, as temperature should then be very slowly but continuously reduced. In that case `itersBetweenTempSteps` can still not be 0. In the SA case one might want the summed `MonteCarloData`. The method can provide that given that a filename is given as input. Furthermore to track the progress of the function one can give a boolean input `showProgress`, which prints the current temperature step. This should not be used in the SA case where `tempsteps` might be enormous

# 4 Results

## 4.1 The Interaction Matrix

**2.1.2** As the interaction matrix is of size 20x20 it is not feasible to include it here in the repport. I have however implemented a validator function which returns boolean `true` if the interaction matrix is of the correct size and is symmetric. That combined with an if sentence at the start of main throwing a `cerr` if the interaction matrix is invalid demonstrates that my code works. There is also a copy of the interaction matrix saved as "interactionMatrix.txt" in both the files for the python implementation and the C++ implementation.

## 4.2 Comparing random structures

**Demonstrates the code created for 2.1.1 by preforming task 2.1.3** In this section the constructor of `ProteinSystem` is ran in random mode, which is detailed in section 3.2. With this i generated protein 1,2 and 3 in figure 1.

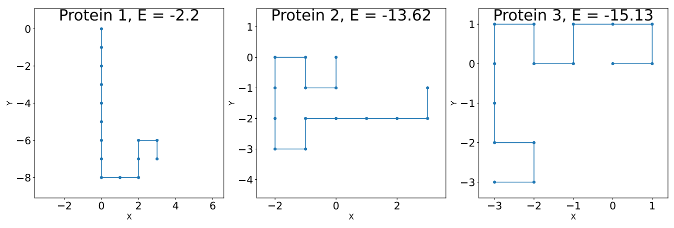

Figure 1: Proteins with their respective energies

This result demonstrates the fact that interaction matrix represent binding energies between any two amino acids. If one estimates each element in the matrix as $-3k_b$, then one would expect the energy of each protein to roughly equate to $-n3k_b$ Where $n$ is the number of nearest neigbour pairs in the protein. By this estimate Protein 1 should have $E = -3k_b$, Protein 2 should have $E = -12k_b$ and protein 3 should have $E = -15k_b$. This energy result corresponds highly with the energies measured by the method function `calculateEnergies`, which is shown in the figure texts.

## 4.3 Simulations with $T \in \{1, 10\}$

**Demonstrates the code for 2.1.4 by preforming 2.1.5. Same for 3D sections 2.2.2, 2.2.1**

The `nMonteCarloSweep` method was used on the same system (with temperature 10) 1, 9 and 90 times giving the positional data of the system after 1, 10 and 100 sweeps. This first protein is shown in figure 2.

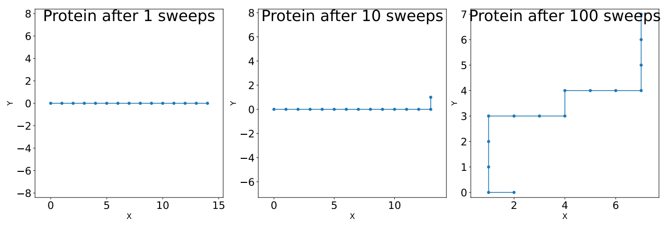

Figure 2: Evolution of the protein

As one can see there is no change after the first sweep. This means effectively that the end points where not drawn. After 10 sweeps the end point has moved, but the movement has not propagated through the protein. This shows that for the first few sweeps when only the end points are allowed to move, the simulation is slow. After 100 sweeps the folding has started to propagate. Note that in all 3 cases the energy of the system is still 0, which partially demonstrates that for such high temperatures most moves are legal, but it also demonstrates that 100 sweeps is not really enough to determine the behaviour of the system at a given temperature. One really has to average over multiple sweeps after the system has settled to get a good result.

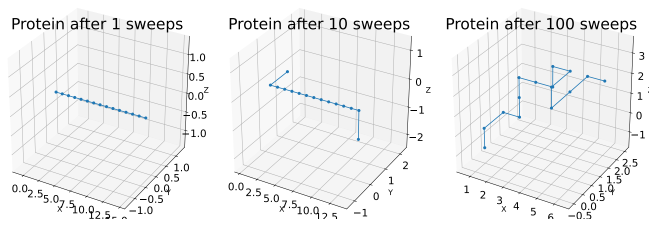

Figure 3: Evolution of the protein

The evolution is largely the same in 3D, as shown in figure 3. The difference is largely in the extra possible configurations. The main difficulty i can see from the 3D solution is that since i have disallowed the crank moves it will take even longer for the system to settle into a correct state corresponding to the temperature. This is because For 3D the corners still only have a single legal move whilst the ends have 5. The movement has to propagate from the end points and now there are even more possible configurations the end points has to generate.

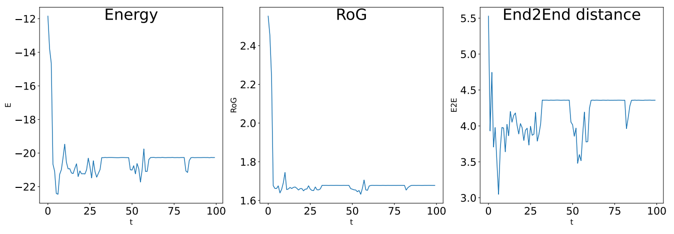

Figure 4: Time averaged Monte Carlo Data

Finally a system at `T` $= 1$ was generated.
For this i have not generated intermediary positional data, instead i want to investigate the resulting `MonteCarloData`. A system was set up and `nMonteCarloSweeps` was ran with $n = 10$ million. The data was taken into python, and blocks of 10000 was averaged together by the `blockAverage` function. The result was figure 4. The term "Time averaged" is slightly misleading. Assuming each sweep takes some "time" one

can then call the units on the x-axis "time". The figure shows that the protein takes a long time to settle into a low energy configuration when it is started at `T` = 1. I will show later an alternative method to get the system to settle faster. From the figure it is clear that 1M sweeps was not enough to put it into equilibrium, however after about 4M the system is definitely in equilibrium. The system has a few fluctuations later, which is a feature of the Monte Carlo method preventing a system from being stuck in a locally minimal energy state. The three quantities are highly correlated. When the system is in equilibrium all the graphs are constant, and when it is not the graphs are chaotic. To determine stable behaviour it seems like Energy and RoG is most useful due to the fact that its behaviour is less erratic then the system is not in equilibrium.

## 4.4 Phase diagrams in 3D & 2D

**2.1.7, 2.2.3** The results in this section was very computationaly expensive. I ran a total of 12 systems on separate threads, 6 for 2D and 6 for 3D. I used the `annealSystem` method with 3M calls to `nMonteCarloSweeps` every temperature step and with 20 temperature steps. This was chosen simply because it was the most my system could handle in a possible timeframe (simulation took 2h even with compiler optimalizations in build mode). From the initial plots of the phases i saw that this was enough to make a smooth phase diagram for $N = 15$ but not for 50 and 100. The systems simply did not have enough steps to allow them to curl up properly. To remedy this problem i slightly cheated by running a second set of simulations (the other 6) where i let the systems be initialized with RNG. That is to say they where not initially stretched out. This made it so the longest proteins got to contract, however my conclusion is that it was still not enough simulations to get a correct phase diagram. I would need to let the simulation run for a few days with even more calls to nMonteCarloSteps. I have still decided to present the phase change data for $N = 50$ and $N = 100$ However i have chosen the randomly initialized runs.

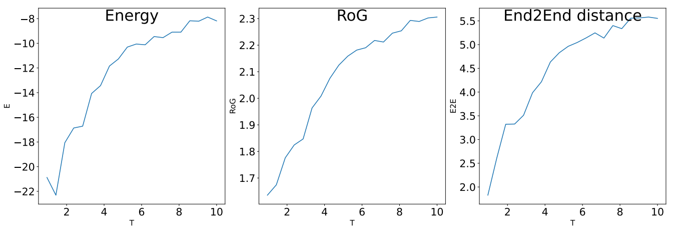

Figure 5: Phase diagram N = 15, 2D

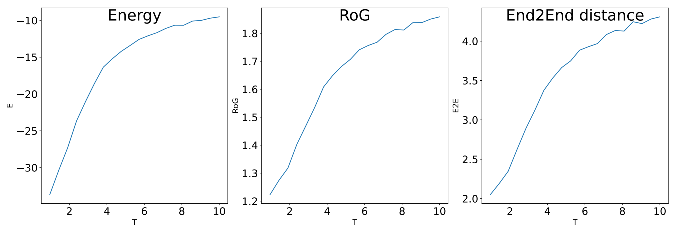

Figure 6: Phase diagram N = 15, 3D

First of, lets consider the phase change diagrams for 2D and 3D with N = 15, as shown in figures 5 and 6 respectively. Here there is quite a clear phase change when the system goes above about 4 degrees, which is interesting, as this is also the maximal energy in the interaction matrix. I would initially expected the critical temperature to lie above the average of the interaction matrix, however it seem this is not the case. Below the critical temperature the binding energy exedes the thermal energy, which result in the breaking of nearest neighbour bonds becoming increasingly unlikely. So the system behaves more fluidly above 4 degrees, and more like a solid below 4 degrees. I was not expecting the 3D plot to be smother than the 2D plot. The exact cause of this i am not sure of, however it may be the case that the extra possible configurations in 3D averages the system out easier in 3D than in 2D.

For the 3D and 2D plots for N = 50 and N = 100, the results are not as pretty. There is some indication of there being two phases, as whith the N = 15 case, however The critical temperatures are not as clear.

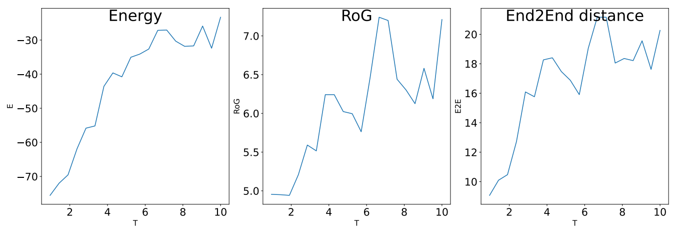

Figure 7: Phase diagram N = 50, 2D

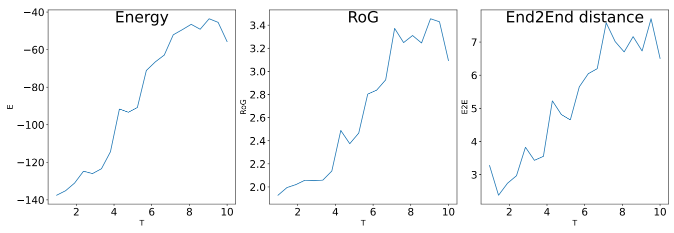

Figure 8: Phase diagram N = 50, 3D

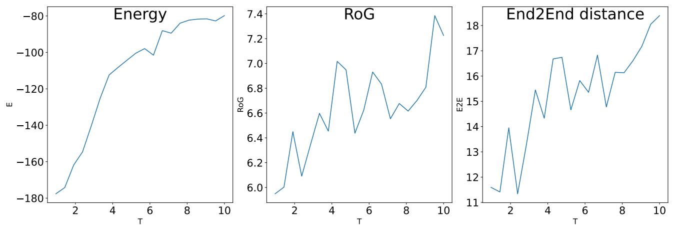
Figure 9: Phase diagram N = 100, 2D

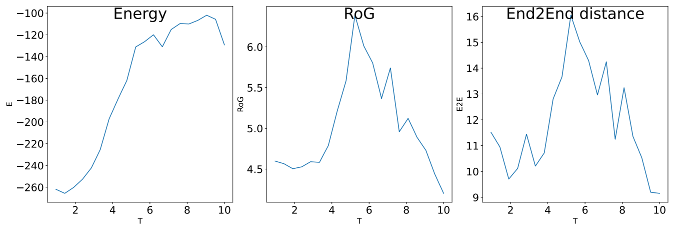
Figure 10: Phase diagram N = 100, 3D

From the phase diagrams one can also see that the 3D cases for higher values of N have much rougher temperature-energy phase diagrams, and the erratic behaviour of RoG and End2End distance indicates that the systems are not really reaching equilibrium. The simulations need more sweeps for a proper phase diagram. The protein structures for the different N are also shown in the python program, and especially for the N = 100 caste those plots show that the N = 100 system form two separate "Curls" at each end. The System does not have the time to curl all the way up. Nevertheless one can, as state din the above paragraph, still see some of the same properties as the N = 15 case had.

## 4.5 Annealing

**2.1.8** This problem was solved in 3 steps. I first preformed `nMonteCarloSteps` twice on two systems at $T = 1$ with the same amino acid types. The `monteCarloData` was saved. Then i used `anealSystem` with a high number of temperature steps, and the `monteCarloData` was also saved. This was done for me to be able to compare the time it takes for the system to reach thermal equilibrium.

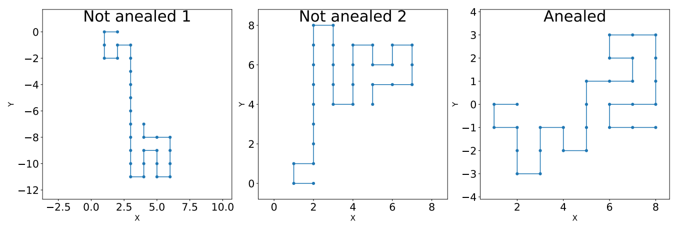
Figure 11: Tertiary structures generated by annealing and by brute force

The simulations where ran such that in all 3 cases there where 10 million Monte Carlo Sweeps. From the Plots it looks like neither of the non-annealed states really are the lowest energy state. The systems have however reached a steady state. This is implied by the figure 12. The brute forced systems have reached a steady state, whilst the anealed system for which the energy is constanly lowered the system reaches a much lower energy state. This would imply that the curling the proteins in the brute forced methods already have done gives it too large binding energy to allow it to split up again to create a lower energy state. Comparing it to a physical example, the protein is like a fountain which has had its water frozen in mid air. It is certainly not the lowest energy state for the ice, but to reach a lower energy state it needs to first melt. In the annealing process it is as if we let the water from the fountain hit the ground before freezing, allowing it to attain a lower energy state.

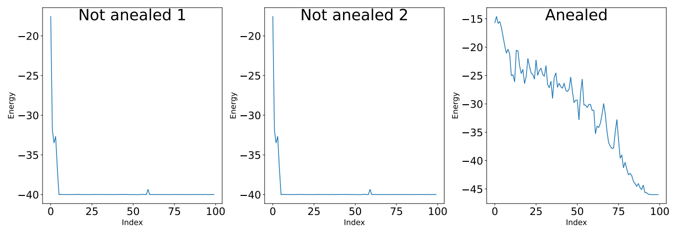
Figure 12: Energy over time of simulation

Now it could still be that by freezing the system even slower one could reach a even lower energy state. It also seems like from the figure that one can probably start the simulation just slightly above the critical temperature. It is the slow freezing of the protein near the critical temperature which allows the system to settle into a low energy state.

## 4.6 Repulsive Interaction Matrix

**2.1.9** In this task i ran the system using `anealSystem` such that `nMonteCarloSweep` was called 50M times. Since i did not have the time to rewrite the plotting scripts in python to show amino acid type with colors i could not investigate if the protein would show any of the repulsive protein interactions. Instead, to generate an interesting structure, i set quite a lot of the elements to be repulsive (in total 6 pairs where 3 where self interactions). I i wanted to do this totally realistic i could assign some charge to each monomer and let the sign of the interaction equal the product of the signs of the charges that would interact. In that case every diagonal element would be repulsive. In this case, I simply chose some of the pairs to have repulsive interactions. The resulting structure is shown in Figure 13.

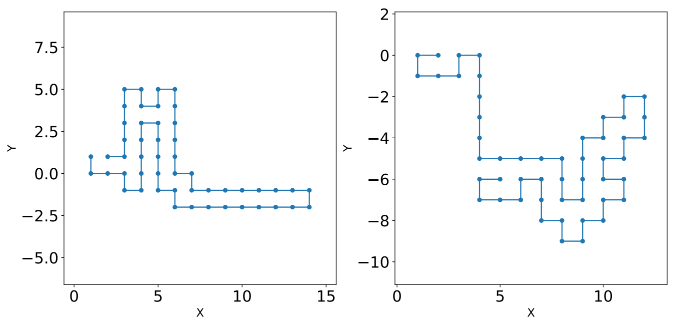

Figure 13: Structure with repulsive interactions

As seen in the figure, the structure appears to be more open and extended compared than i would expect for a interaction matrix with only negative elements. In that case i would predict that the system would tend towards curling up on itself, as that allows for the most nearest neighbours. In this case however i annealed it extremely slowly and the structure is still stretched out, indicating that the repulsive forces are preventing them for curling up completely.

## 5 Conclucion

In conclusion, the implementation of the Protein Folding Model in both 2D and 3D was successful. The model accurately represented the interaction matrix and generated structures with energies consistent with the expected values based on nearest neighbor interactions. The Monte Carlo simulations with various temperatures allowed us to observe the protein folding process and analyze the differences in behavior at different temperatures. Furthermore, the annealing approach demonstrated its efficiency in finding lower energy states compared to brute force methods.

The phase diagrams for both 2D and 3D provided valuable insight into the protein folding behavior at different temperatures and chain lengths. The phase change was apparent for N=15, while for longer chains, more extensive simulations would be required to obtain accurate phase diagrams. The introduction of repulsive interactions in the interaction matrix led to more open and extended structures, highlighting the importance of considering such interactions in protein folding.

Future work could include implementing crankshaft moves in 3D, improving the efficiency of the simulations by using parallelization or other optimization techniques, and investigating the effects of more realistic interaction matrices with charge-based repulsive interactions. Additionally, extending the model to incorporate more biologically relevant features, such as the explicit representation of solvent molecules, could provide further insights into the protein folding process.

## References

[1] Python Software Foundation. *Numeric Types — int, float, complex*. Version 3.10.0. Accessed: 2023-05-04. Python Software Foundation. 2021. URL: `https://docs.python.org/3/library/stdtypes.html#numeric-types-int-float-complex`.