Assignment 2 – Atomic-Level Molecular Modeling

CS/BIOE/CME/BIOPHYS/BIOMEDIN 279

Due: **Thursday, October 31, 2024 at 1:00 PM**

The goal of this assignment is to understand the biological and computational aspects of molecular energy functions and of predicting the three-dimensional structure of a folded protein.

1 Preliminaries

1. If you are working on your local computer, **make sure you can run PyMOL 2.5 and Python 3.9**. As mentioned in the software handout, we heavily recommend that you have Anaconda for Python 3.9 installed as well.
2. Please start your assignment early so that the course staff can help you troubleshoot any issues in office hours or Ed.
3. **Please save your code from the "Exercises" listed below**. We'll ask you to append the code you wrote for the Exercises at the end of your written submission.
4. In Part 4, you will be running a Colab notebook. Historically, students have run into issues on Safari. We have confirmed that the Colab runs successfully in Firefox, Edge, and Chrome.

2 Force Fields and Free Energy Calculation

2.1 Potential Energy and Force Fields

Molecules, including proteins and other biomolecules, prefer to occupy energetically stable (i.e. low-energy) three-dimensional structures. Thus, in order to predict which conformations a protein or other biomolecule is likely to adopt, it's important to understand the forces acting between its atoms.

One frequently used type of potential energy function for biomolecules, called a molecular mechanics force field, is typically defined as a sum of bond length, bond angle, dihedral angle, electrostatics, and van der Waals terms (see slides from the lecture on "Energy Functions & their Relationship to Molecular Conformation," particularly slide 24). Let's define an individual conformation c of a biomolecular system as a particular arrangement of all the atoms that is specified by precise coordinates for each atom. Then, the potential energy associated with conformation c (defined as $U(c)$) can be modeled as

$$U(c) = U_b(c) + U_\theta(c) + U_\phi(c) + U_e(c) + U_v(c)$$

Note: $U_b(c)$: bond length stretching; $U_\theta(c)$: bond angle bending; $U_\phi(c)$: torsional angle twisting; $U_e(c)$: electrostatic interactions; $U_v(c)$: van der Waals interactions.

Question 1: *For a potential energy function defined by such a molecular mechanics force field, which two of the following terms will generally require the most computational time to evaluate, in aggregate: bond length, bond angle, torsional angle, electrostatics, and van der Waals? (Assume the molecular system being simulated contains a very large number of atoms.)*

2.2 Conformational States

Recall that proteins and other biomolecules are constantly changing shape as their atoms move around. Thus, when we talk about protein structure prediction, we're looking to find a set of similar conformations in which the protein will spend most of its time.

Like in question 1, let c represent an individual protein conformation. We also define a set C of conformations similar to c. We refer to such a set of conformations as a **conformational state**.

We know that the probability that a biomolecule is in some conformation c is Boltzmann distributed according to $U(c)$, where $U(c)$ is the potential energy associated with conformation c. This means we can express the total probability that the biomolecule will adopt some conformation in the conformational state C by summing the probabilities of adopting each conformation in the conformational state.

$$\begin{aligned} \Pr(c \in C) &= \frac{1}{Z} \sum_{c \in C} e^{-U(c)/(k_B \cdot T)} \\ &\propto \sum_{c \in C} e^{-U(c)/(k_B \cdot T)} \end{aligned} \tag{1}$$

k_B: *Boltzmann constant*
T: *temperature*

Z is a normalizing constant such that the summed probability across all possible conformations is 1. If we're interested in relative probabilities of conformational states, we don't need to calculate Z.

Intuitively, a conformational state is well-populated if the energy of each conformation in the conformational state is sufficiently low and the number of conformations in the conformational state is sufficiently large.

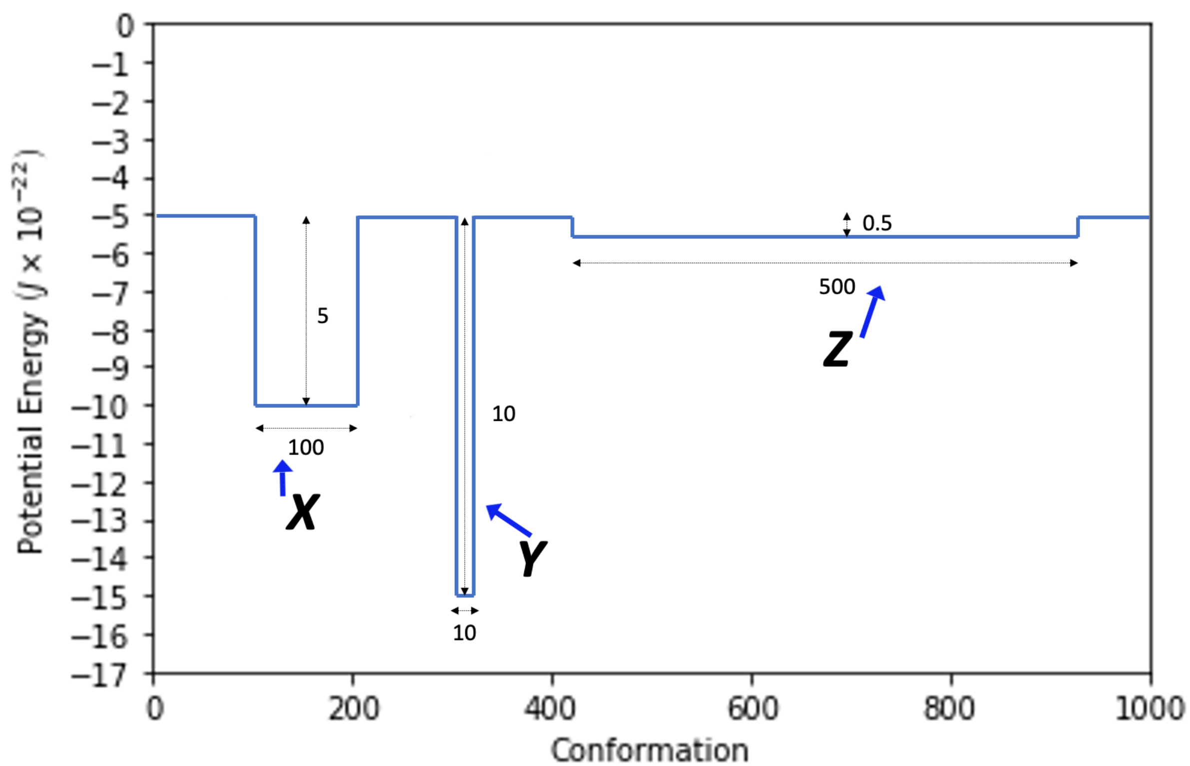


Figure 1: Potential Energy vs. Conformation, labeled.

.

Question 2: *Consider Figure 1. Let's say this plot represents the potential energy of a given protein (and the water surrounding it) as it adopts different conformations.*

(a) Without doing arithmetic (calculations), which conformational state (X, Y, or Z) would you expect to be preferred at extremely low temperatures? Which conformational state (X, Y, or Z) would you expect to be preferred at extremely high temperatures? Why? Hint: consider the number of conformations in each conformational state, and the potential energies of the conformations in each conformational state.

(b) Now, calculate the approximate relative percentage present of each conformational state at each of the temperatures: 1 K, 100 K, and 310 K by using Equation (1). Display your results in a table as shown below. Note that 310 K (equivalent to 37 °C or 98.6 °F) is normal human body temperature.

	Percentage Present of each Conformational State (%)		
Temperature (K)	**X**	**Y**	**Z**
1			
100			
310			

Hints: $k_B = 1.38 \times 10^{-23}\text{J/K}$. *You may ignore the time the protein spends outside of conformational states X, Y, and Z. Pay close attention to the sign of the potential energy.*

This question should illustrate that the structure of a protein (or any molecule for that matter) varies with temperature.

3 Deriving a knowledge-based energy function

In this problem, we will build a "knowledge-based" energy function. Instead of approximating the potential energy of atomic interactions and then integrating these to compute free energies, this type of energy function directly estimates the free energy of a candidate structural model by incorporating knowledge of experimentally determined protein structures. Candidate structural models are deemed lower energy (more likely) if they share properties with structures of other proteins. The energy function we will implement is inspired by the paper *Statistical potential for assessment and prediction of protein structures* by M.-Y. Shen and A. Sali, which has accrued over 2000 citations.

Our knowledge-based energy function will be based on distances between pairs of residues. We will record the distances between residues in a training set of protein structures from the PDB and give favorable scores to candidate structural models that include inter-residue distances that are observed more frequently than would be expected by random chance. Each pair of residue types (e.g. serine and tryptophan), will have its distances recorded separately. In this way, the energy function will be able to distinguish which types of residues it is favorable to position closely.

Specifically, the overall energy for a candidate structural model will be a sum over energy values for

each pair of residues.

$$G = \sum_{i=1}^{N} \sum_{j=i+1}^{N} g(d_{ij}, A_i, A_j) \tag{2}$$

where N is the number of residues in the protein, d_{ij} is the distance between residues i and j, A_i and A_j are the amino acid types of residues i and j, respectively, and g is a per-residue-pair energy function. We'll go into more detail about the per-residue-pair energy function later.

We provide starter code in "statistical_potential.py". As you complete the exercises below, you should only edit the code in places between the "############" lines. We've provided unit tests so that you can check the correctness of your code as you implement the components. To run the unit tests, open a terminal window and navigate to your Assignment 2 directory. You can then run the unit tests using the following command.

```
> python test_statistical_potential.py -v
```

Note: You may have to replace `python` with `python3` in order to point to the correct python version. If you are on the LTS machines, you may need to type `python3.9`.

There are some additional notes in the Appendix of this document which explain the overall structure of the Python file and the role of unit tests. There are also comments in the starter code, which should help with the implementation of the functions you will write.

3.1 Distribution of inter-residue distances in structures from the PDB

First, we will compute distributions (histograms) of the distances between pairs of residues in a large set of structures from the PDB. We will compute a separate distribution for distances between each pair of residue types (i.e., one for distances between aspartate and alanine residues, one for distances between aspartate and arginine residues, etc.).

In the code, we will represent these distributions as a list of counts for various distance bins. The ith entry in the list will store the count for distances between i and $i + 1$ angstroms (Å). For example, the first entry will correspond to the number of occurrences between 0 and 1 Å, the second entry will correspond to the number of occurrences between 1 and 2 Å, and so on. These lists will each have 15 entries, and we will ignore residue pairs with distances over 15 Å.

Exercise 1: *In "statistical_potential.py" implement the function "observed_distribution". Your code should pass the tests prefixed with "test_observed_distribution" in "test_statistical_potential.py" after implementing this (the first 3 unit tests). Plot the observed distribution for aspartate and arginine by running the below command.*

```
> python statistical_potential.py ASP ARG
```

***Note**: While distributions are usually shown as bar charts (histograms), this assignment displays distributions as line graphs to clearly show the difference between the observed and expected distributions.*

3.2 Accounting for the expected distribution of inter-residue distances

Given that aspartate and arginine residues typically have opposite charges and presumably interact favorably with one another, it might seem a bit surprising to see that there are many more occurrences of these residues far apart than close together. But is this really unexpected?

We will now consider how many occurrences one would expect in each distance bin if all the residues were positioned randomly. The expected number of occurrences of residue distances between distance 0 and distance d_{ij} is proportional to the volume of a sphere centered at a given point with radius d_{ij}. Therefore, calculating the expected number of occurrences between distances d_{ij} and $d_{ij} + 1$ is equivalent to asking how much volume is located at distances between d_{ij} and $d_{ij} + 1$ from a given point. **Note**: The expected number of occurrences between distance 0 and distance d_{ij} is proportional to the volume of a sphere with radius d_{ij} because the distances are measured in 3 dimensions (x, y, and z).

Two important notes: (1) The expected number of occurrences of distances is influenced by the overall number of each residue type in the protein. You do not need to include this in your solution. Elsewhere in the code, we will normalize the entire expected distribution so that the expected and observed counts are the same at 15 Å. (2) In the Shen and Sali paper, they consider a very complex expected count function that accounts for the fact that at distances approaching the size of the protein the expected counts decrease. Do not attempt to model this effect for the purposes of this assignment.

> **Exercise 2:** *Implement the function "expected_ distribution". Your code should pass all the test cases from Exercise 1 as well as the "test_ expected_ distribution" test case after implementing this (so it passes the first four of seven test cases). Hint: the volume of a sphere with radius r is $\frac{4}{3}\pi r^3$*

Run, the code again for aspartate and arginine. You should now see that there are more occurrences of distances in the 4 to 10 Å range than we would expect by chance.

> **Question 3:** *In the plot for aspartate and arginine, you should see that there are fewer observed than expected occurrences for very short distances (0–3 Å). We will soon see that this is the case for every pair of residue types. What physical effect causes this?*
> *Hint: How wide is an atom?*

3.3 Deriving residue-pair energies

We will now combine the observed and expected distributions we computed above into the per-residue-pair energy function, g. Specifically, the energies will be given by the negative log ratio of the two distributions.

$$g(d_{ij}, A_i, A_j) = -\log \frac{N_{obs}(d_{ij}, A_i, A_j)}{N_{exp}(d_{ij}, A_i, A_j)} \quad (3)$$

Here $N_{obs}(d_{ij}, A_i, A_j)$ is the number of occurrences of residue types A_i and A_j at a distance of d_{ij} in the set of solved protein structures and $N_{exp}(d_{ij}, A_i, A_j)$ gives the corresponding expected count. **Note**: log denotes $\log_e$, which is also known as the natural log (ln).

This way of combining the two distributions favors distances that are observed more often than

expected and results in terms that have the mathematical properties of energies. Following the convention for energies, distances that are more favorable should get lower scores. Distance bins with greater observed than expected counts should also get negative scores.

Exercise 3: *Implement the function "distributions_to_energy" in the code. Your code should now pass the "test_distributions_to_energy" test, in addition to all the tests from the previous questions (so it passes the first 5 test cases).*

We will now run the code for several pairs of residue types and examine the learned per-residue-pair energy function. Remember, to run the code on, for example, aspartate (ASP) and glutamate (GLU), you can use the command:

```
> python statistical_potential.py ASP GLU
```

Question 4:

(a) Run your code for aspartate (ASP) and arginine (ARG), and then aspartate (ASP) and glutamate (GLU). What physical interaction explains the difference in energies?

(b) Run your code for leucine (LEU) and isoleucine (ILE). What "effect" causes these residues to interact favorably?

(c) Run your code for cysteine (CYS) and cysteine (CYS). What special property of cysteine explains the favorability of positioning cysteine residues very close to one another?

***Hint**: Slide 42 in "Biomolecular structure (including protein structure)" contains examples of amino acid properties.*

Exercise 4: *Compute per-residue-pair energy functions for all pairs of residue types by running*

```
> python statistical_potential.py compute
```

This will plot all of the energy functions and write them to a file called "energies.txt" in the "starter_code" directory. You don't have to submit the "energies.txt" file for this exercise, but "energies.txt" is required for the upcoming questions. Test cases are not relevant for this question.

3.4 Protein structure prediction

Now that we have code to compute per-residue-pair energies, we can sum them all to get scores for entire proteins. We will use these scores to rank candidate structural models for a set of 20 proteins. For each protein, we've provided 300 candidate models, each of which is labeled with an RMSD to an experimentally determined structure (the "native structure"). Ideally, our scoring function will assign the most favorable (lowest) score to models similar to the native structure.

Exercise 5: *Implement the "scorer" function. This function computes the energy for a single structural model. Your code should now pass all seven test cases.*

After your code passes the test cases, use the below command to run the scoring for all of the test proteins.

```
> python statistical_potential.py score
```

For each protein, the code will print 1) the RMSD of the structural model selected by the energy function, 2) the lowest RMSD model across all candidate models, and 3) the median RMSD of the candidate models. You should see that the energy function performs much better than the median RMSD (which is what would be expected if we picked a random structure), but that the the model by no means has perfect performance. For each protein, the code will also produce a scatter plot of RMSD vs. energy for each candidate structural model.

Question 5:

(a) For what fraction of the proteins does the energy function predict a close to native candidate structural model, where "close to native" is defined as having an RMSD of less than 4.0 Å?

(b) Include the scatter plots of RMSD vs. energy in your write-up.

(c) Such scatter plots are commonly included in publications describing knowledge-based energy functions. What relationship between the RMSDs and energies is needed such that one can confidently select a near-native candidate structural model based on the energy function? A simple answer is sufficient.

(d) Here we are using the energy function to score candidate structural models generated with other software. Energy functions are also often used to guide generation of candidate models ("sampling"). As you saw in lecture, sampling algorithms generally permit transitions that increase the predicted energy. Based on inspection of the scatter plots for the proteins with PDB codes 1bbh and 1cid, why is this essential to arrive at a near-native structural model? ***Hint****: Does the energy increase monotonically with the RMSD in these plots?*

Let's visualize a case where our energy function makes a correct prediction. Open PyMOL and navigate to the assignment directory using the "cd" command. Let's load the "native" (experimentally determined) structure, a "correct" (close to native) structural model, and an incorrect structural model.

```
PyMOL> load test-pdbs-moulder/1mdc/native.pdb
PyMOL> load test-pdbs-moulder/1mdc/model2.pdb, incorrect
PyMOL> load test-pdbs-moulder/1mdc/model4.pdb, correct
```

Then, align both of the candidate structural models to the native structure (ex: `align model4.pdb, native.pdb` in PyMOL). You should see that the "correct" model has a much lower RMSD than the "incorrect" model.

We've provided a PyMOL script "visualize.py" to show residue pairs that are assigned particularly favorable per-residue-pair energies by our energy function. (Note that this script depends on "energies.txt" to get the energies, so you must have computed the energies before running this script.) Running the below commands will add lines between residue pairs with highly favorable (less than −0.75) per-residue-pair energy in the structural models.

```
PyMOL> run visualize.py
PyMOL> show_energies correct
PyMOL> show_energies incorrect
```

Question 6: *A key region of the protein that is correctly predicted in the correct structural model, but incorrectly predicted in the wrong structural model, involves the residues in the vicinity of residue 18. In the correct structural model, interactions between several pairs of residues in this region are deemed to be highly favorable by the energy function. What "effect" causes these interactions to be highly favorable? What is surrounding residue 18 in the incorrect model that is likely energetically unfavorable?*
Hint: *A quick way to locate residue 18 is to use the command "select resid 18".*

4 Protein structure prediction with AlphaFold

Now we will do structure prediction using AlphaFold (specifically, AlphaFold 2). Like the scoring function that we just created, AlphaFold is a knowledge-based method, in that it was trained to reproduce experimentally-determined structures of proteins, as opposed to directly capturing the underlying physical laws. However, there are several fundamental differences between the knowledge-based potential we just created and AlphaFold that enabled AlphaFold's unprecedented success.

First, AlphaFold does not just score protein conformations, but actually directly outputs 3D structures.

Second, our knowledge-based potential started with a pre-specified functional form, which introduced a variety of constraints on the final scoring function, whereas AlphaFold uses a deep learning approach that can learn far more expressive functions.

Question 7: *Name one assumption that we made in constructing our knowledge-based energy function that limits its ability to represent complex scoring functions.* ***Note***: *There are multiple possible answers, and we're not looking for one specific answer.*

Finally, AlphaFold is able to leverage two types of widely available experimental data that can aid in structure prediction: sequences of related proteins and experimentally determined structures of related proteins. To gain insight into how AlphaFold leverages these data types, we will predict the structure of flavin reductase using AlphaFold with different combinations of input data.

We will use a Google Colab notebook version of AlphaFold called ColabFold. The notebook contains helpful usage information. As mentioned in the Preliminaries section, we have tested and recommend running the Colab in Firefox, Chrome, or Edge.

As a baseline, let's use AlphaFold to predict the structure of flavin reductase using only its sequence. First, we will need to look up the amino acid sequence. Amino acid sequences for most characterized proteins are available from UniProt, a useful resource that organizes a large amount of information about each protein (`https://www.uniprot.org/`).

Here is the entry for flavin reductase. You can get the sequence by navigating to the "Sequence" section and clicking "Copy sequence".

Function
Names & Taxonomy
Subcellular Location
Disease & Variants
PTM/Processing
Expression
Interaction
Structure
Family & Domains
Sequence
Similar Proteins

PS00237 G_PROTEIN_RECEP_F1_
PS50262 G_PROTEIN_RECEP_F1_

Protein family/group databases

GPCRDB | Search...

Sequence

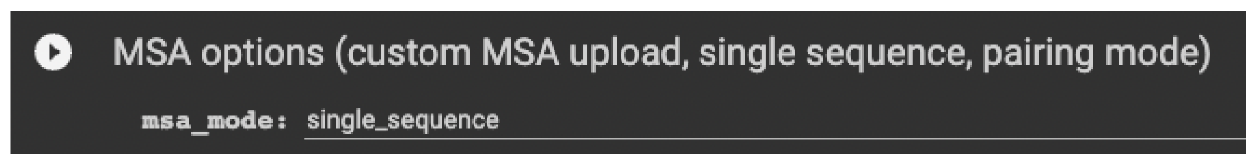
Tools Download Add Highlight Copy sequence

Length 361
Mass (Da) 41,354

MNESRWTEWR ILNMSSGIVN VSERHSCPLG FGHYSVVDVC

CLACISVDRY LAITKPLSYN QLVTPCRLRI CIILIWIYSC

Now, let’s setup the ColabFold run. Paste the flavin reductase sequence into the “query_sequence" text box. Then, to disable the use of related sequences (multiple sequence alignments), set “msa_mode” to “single_sequence”.



To run the prediction in ColabFold, navigate to "Runtime" in the Colab toolbar and select "Run all". The results should automatically be downloaded as a .zip file. Unzip the file and visualize the top-ranked model using PyMOL. ColabFold predicts 5 potential models, which correspond to 5 .pdb files. We will focus on only the top-ranked model, which includes “rank_001” in the file name. Compare the predicted structure to the experimentally determined structure of flavin reductase with PDB ID: 1HDO. You can download this structure using the fetch command in PyMOL.

Question 8: *Align the predicted and solved structures to compare them. What is the RMSD between the predicted and experimentally determined structures? Do you notice any differences between the predicted structure and the experimental structure?*
Note*: We are not looking for one specific answer. We expect a general comparison of the two structures.*

In addition to the sequence of the query protein, AlphaFold can make use of information about the sequences of related proteins. Before being input to the model, these sequences are aligned together to form a “multiple sequence alignment” (MSA).

Let’s make a prediction for the same protein, but this time using MSA information. Ensure that the flavin reductase sequence is pasted into the “query_sequence" textbox. Set the “msa_mode” to `mmseqs2_uniref_env` (“MMSeqs2: (UniRef+Environmental)”) and run the prediction again. You can re-run all of the cells after updating the options by clicking “Runtime” - “Restart and run all”.

Question 9: *Again, download the results and compare the "rank_001" model to the experimentally determined structure. What is the RMSD between the predicted and experimentally determined structures? Do you notice any differences between the predicted structure and the experimental structure?*
Note: *We are not looking for one specific answer. We expect a general comparison of the two structures.*

If you like, you can view the sequences used to make the prediction by opening the .a3m file in a text editor.

AlphaFold is also able to make use of experimentally determined structures of related proteins.

AlphaFold has an automated pipeline for fetching template structures from the PDB, but for the sake of interpretability, let's provide a template manually. In PyMOL fetch PDB ID: 4R1S. The protein in 4R1S, which is from petunias (the plant), is similar in sequence to flavin reductase and carries out a similar function. (Note that UniProt uses a different naming system than the PDB. UniProt calls this protein CCR1_PETHY while the PDB calls the structure 4R1S). In addition to loading the structure into the PyMOL window, this will produce a file "4r1s.cif". We will have AlphaFold use this model as a template by setting " template_mode" to "custom". We'll use the same flavin reductase sequence and the same "msa_mode" as the previous question, so paste the flavin reductase sequence into the "query_sequence" and set the "msa_mode" to `mmseqs2_uniref_env`. Starting the prediction by clicking "Runtime" and then "Restart and run all" will result in a "Browse" button appearing. Click it and select the file "4r1s.cif".

Question 10: *Again, download the results and compare the "rank_001" model to the experimentally-solved structure. What is the RMSD between the predicted and experimentally determined structures? Do you notice any differences between the predicted structure and the experimental structure?*
Note: *We are not looking for one specific answer. We expect a general comparison of the two structures.*

Question 11: *Compare the template structure (4r1s) to the experimentally determined structure of flavin reductase. Based on what you see, why are the structures of homologous proteins useful for protein structure prediction? Phrase your answer in terms of the "sequence determines structure, structure determines function" paradigm.*

We note that, while either the multiple sequence information or the templates are sufficient here to give a correct prediction, there are some proteins for which only one or the other source of information is available (and in some especially challenging cases, both types of information are required to make an accurate prediction).

5 Feedback

Again, we'd love some feedback on the assignment! You will receive full credit for this portion for providing any response. Please fill out the assignment feedback form and provide the code word below. We encourage constructive criticism.

Question 12: *What is the feedback form code word?*

6 Challenge Question

This question is intentionally more challenging than the rest of the homework. It is **not required**, and you can receive full credit for the assignment without attempting it. We will never deduct points for a challenge question submission and will award extra credit depending on the quality of your solution.

Question 13: *The challenge question for this assignment will be posted separately on the class website, and an announcement will be made on Ed once it is posted.*

7 Submission Instructions

Similar to Assignment 1, please submit this homework on Gradescope (https://www.gradescope.com/). Type up your answers in the text editor of your choice (LaTeX, Word, etc.), making sure to clearly note the question number associated with your answer. When submitting, make sure to tag your answers with the correct questions on Gradescope.

Copy the code for your implementation of `statistical_potential.py` from all the exercises to the end of your writeup. You only need to include the functions implemented in the exercises: `observed_distribution()`, `expected_distribution()`, `distributions_to_energy()`, and `scorer()`. Then convert the entire document to a pdf. Go to Gradescope, navigate to CS279, "Assignments", then select **Assignment 2**.

If you completed the challenge question, please carefully read and follow the submission instructions at the bottom of the challenge question document.

If you have any issues submitting, please let the course staff know via Ed.

Appendix: Additional Notes on Coding

The Python file "statistical_potential.py" has a lot of moving parts that may make it feel daunting to someone without a ton of coding experience, but never fear! This guide should help you out. Before we dive in, here are definitions of specific terms and how items are formatted in this guide:

- In this section, we use the words "Python file" or "file" to describe the entire contents of "statistical_potential.py".
- Command line arguments are usually shown in typewriter text and underlined (like `this`). The exception is when we show a command line argument within part of a larger command being run, in which case we will use gray highlighting to be consistent with the rest of the document.
- We use the word "mode" to describe one of three types of behavior that the file will execute, depending on which command line arguments are given when running the file. For example, running `python statistical_potential.py compute` will execute a different mode of functionality versus running `python statistical_potential.py scorer`.
- The word "function" is used to designate a block of code with inputs and outputs that works as part of the total file. Names of specific functions are bolded.
- Specific variables in the code are marked with italics.
- We use the term "residue type" in this section to refer to one of the twenty residues found in proteins (for example, aspartate).
- We will use "specific residue" or "residue" to describe a particular instance of a residue (for example, ASP140 refers to a specific aspartate which is in the 140th position of a protein chain).

A Overview

"statistical_potential.py" has three different modes of functionality:

- Mode 1: The user supplies two residue types (for example, `ASP ARG`). The Python file generates distance distributions for all specific residue interactions of those two residue types. This mode is run by executing `python statistical_potential.py ASP ARG`
- Mode 2: The user supplies the word `compute`. The Python file will compute energies for all interaction types in a structure. This mode is executed by running `python statistical_potential.py compute`
- Mode 3: The user supplies the word `scorer`. The Python file will score all interactions according to the energy values generated with the `compute` command when running Mode 2. This mode is run by executing `python statistical_potential.py scorer`

B Mode 1: Energy Distributions

The first three exercises of this assignment involve implementing Mode 1 of "statistical_potential.py". The end result will be a plot with two subplots. One subplot shows the observed and the expected distance distributions of all specific residue interactions between two residue types. The second subplot shows energy as a function of distance for the same interactions. The workflow of Mode 1 is as follows:

1. Input: The user inputs names of two residue types on the command line when running the Python file: `python statistical_potential.py ASP ARG`. The names of the two types are saved as *resname1* and *resname2*.

2. **pair_energy**: The function **pair_energy**, which has already been written, takes *resname1* and *resname2* as input. This function will call a few other functions, as explained below.

3. **read_train_set**: The first line of **pair_energy** runs the **read_train_set** function (also already written). A couple of other functions will run behind the scenes, but you don't need to worry about the details. The most important thing to know is the structure of the output, which is saved under the variable name *train_structures*. *train_structures* contains a list of structures. Each structure is made up of a list of information on each residue: (resname, atomname, resnum, (x, y, z)). resname is the name of the residue type, atomname is the name of a specific atom in the residue, resnum is the residue number, and (x,y,z) are the coordinates.

4. **observed_distribution**: *resname1*, *resname2*, and *train_structures* are input into the **observed_distribution** function. You will implement this function in Exercise 1. For each structure in *train_structures*, you will calculate the distances between every instance of the two residue types. This function will produce the variables x and *y_obs*, where x is a list of distances and *y_obs* is the number of occurrences for a given distance value. Tip: try printing *train_structures* to see the format of the list of structures. Also make sure to use some of the pre-written functions in the file to implement **observed_distribution**.

5. **expected_distribution**: Next, the function **expected_distribution** is run. You will implement this function in Exercise 2. This function takes in the variable x from the previous step and outputs the variable *y_exp*.

6. **distributions_to_energy**: The three variables *x, y_obs,* and *y_exp* are fed into **distributions_to_energy**. You will implement this function in Exercise 3. The output *energy* is a list of energies.

7. Output of **pair_energy**: Up to this point, we have still been working within the **pair_energy** function. We now have four variables of interest: *x, y_obs, y_exp,* and *energy.* These four variables are the output of **pair_energy** and are fed back to the main part of the file.

8. **plot_pair**: This function plots the four variables.

9. Output: The output of the entire Python file in Mode 1 is the plot containing a subplot of distance distributions and a subplot of energy as a function of distance.

The workflow is summarized in the flow chart on the next page:

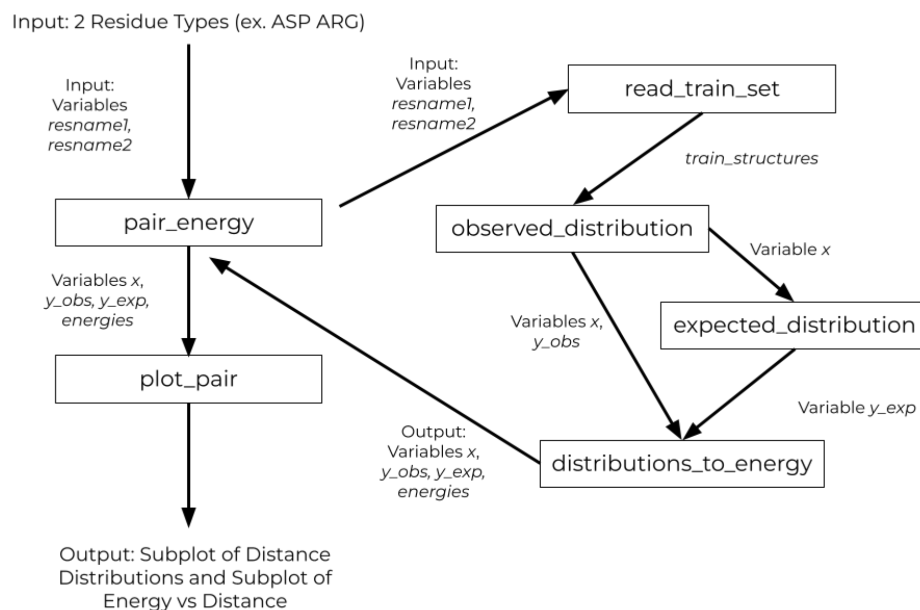


Figure 2: Flow Chart for Mode 1. Functions are shown in boxes. Inputs and output variables are shown in italics. You will code **observed_distribution** in Exercise 1, **expected_distribution** in Exercise 2, and **distributions_to_energy** in Exercise 3

.

C Mode 2: Computing Energy Values

Next, we will talk about Mode 2 of the Python file. Thankfully, this part is very simple. Exercise 4 involves running the `compute` command when executing "statistical_potential.py": `python statistical_potential.py compute`. You do not need to do any coding for this part. Just run the command described in Exercise 4. The output text file "energies.txt" will be used in Exercise 5.

D Mode 3: Scoring Energy Values for a Structure

Mode 3 of "statistical_potential.py" is run with the `scorer` command: `python statistical_potential.py scorer`. Here, you will write a function that takes in a structure and "energies.txt". "energies.txt" contains the energy values of all pairwise interactions between residue types as a function of distance between those residues. You will calculate the distances and then the energy values for these pairwise interactions. The workflow for writing the **scorer** function is outlined below:

1. Input: The user supplies the word `scorer` in the command line, telling the file to execute the scoring workflow.

2. **read_energies**: The pre-written function **read_energies** will read the "energies.txt" file from Exercise 4 and save it as a dictionary under the variable name *energies*. I recommend looking at this file when working on Exercise 5 to understand what information is inside. Each key in the dictionary is a tuple containing two residue type names (ex. "(ASP, ARG)"). Each corresponding dictionary value is a list of energy values. The energy values are given as a discrete function of distance. For example, the first energy value in a given list corresponds to the energy between 0 and 1 Angstroms (a unit of measurement at the atomic scale, equal to 10^{-10} meters), the second energy value is the energy between 1 and 2 Angstroms, and so on. The position of an item in a list is called its index.

3. **score**: The *energies* dictionary is fed into the **score** function. This function will also load a list of protein structures. In this list, there are a few different specific proteins, and several possible structures for each specific protein. Each possible structure has a corresponding structure file. The **score** function will run the **scorer** function for each of these structure files, with the *energies* dictionary and the structure file as inputs. You will code the **scorer** function in Exercise 5.

4. **scorer**: The **scorer** function takes in the *energies* dictionary and a structure file. Your code should calculate the distance between every specific pair of residues in a structure, then "look up" the energy value for that distance given the residue type of those two residues. This "looking up" requires working with both dictionaries and indexing. Here are some resources that explain how to use dictionaries and indexing. Some functions already written within the code should help with the distance calculation.

5. Output of **scorer**: The output of your **scorer** function goes back to the **score** function as a variable which is also called *score*.

6. Output of **score**: The rest of the **score** function runs. This will generate plots of energy scores for all structures of a specific protein as a function of RMSD from the native structure of that protein. All of these plots will be output into the same single image.

7. Output: The output of Mode 3 is the image described in Step 6.

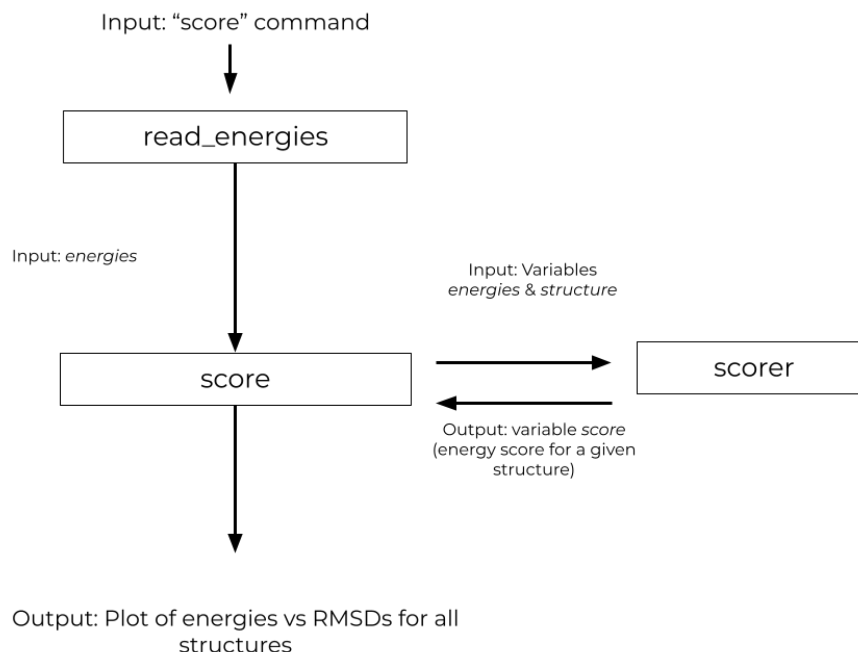


Figure 3: Flow Chart for Mode 3. Functions are shown in boxes. Inputs and output variables are shown in italics. You will code **scorer** in Exercise 5

.

E A Note on Unit Tests

For many people, this class may be their first exposure to the idea of unit tests. A *unit test*, or test case, is a scenario written to ensure that a piece of code functions the way it's supposed to. A unit test will output a message that the code worked if it functioned properly, or will give an error message if it didn't. In practice, developers usually write their own unit tests to check their code, but here we've written them for you. As mentioned above, running

```
> python test_statistical_potential.py -v
```

will execute the unit tests. There are seven tests for this assignment. If all seven tests pass, the file will output `"OK"`. Otherwise, it will output the number of failures, along with any error messages in the code. In this assignment, we will build functions one at a time, so you will not be able to pass all unit tests right away. The instructions for each exercise (except Exercise 4) will tell you how many unit tests you should be able to pass at that point in the assignment.