# Final PR: Protein Folding

CS 182/282 Spring 2024

**Released:** Friday, May 10th, 2024

**Due:** 11:59pm on Friday, May 17th, 2024

## Overview

Protein folding is a critical task in the world of biology. Not only is it useful for discovering new drugs that might save lives across the globe, but it is also useful for studying molecular dynamics, elucidating cellular pathways, and validating gene annotations to improve our understanding of molecular biology as a whole.

When a protein sequence is first transcribed, it starts out in a relatively random conformation (geometric position). However, in one of the most beautiful processes biology has to offer, this transcribed amino acid sequence rapidly folds into a specific, repeatable conformation. In particular, hydrophobic interactions (attractive forces between hydrophobic residues) quickly drive the transcribed sequence close to this optimal conformation, while more subtle interactions push the protein to its final conformation after this hydrophobic collapse.

However, protein folding is an extremely difficult computational task. Even with many simplifications, such as folding proteins only in 2D and measuring a fold only on its hydrophobic interactions (only worrying about hydrophobic collapse), protein folding frameworks are often NP-complete. As a result, the study of protein folding has headed towards more heuristic algorithms and approaches, such as deep learning. In this project, we'll focus on folding proteins on a 2D square lattice to get a sense for the difficulty of the problem.

To navigate this document, start in the introduction (which has some basic details on the framework we're using) and then read the roadmap to get a sense for the broad strokes of the project. Each step in the roadmap will be elaborated on significantly in the following sections that you can refer to as you program. After these sections, you will find a few conceptual questions that you will submit in a document called `theory.pdf` as well as a specification for the program you will write.

## Introduction

First, let's formalize what we mean by a 2D lattice and hydrophobic contacts. For our purposes, a 2D lattice is essentially a grid over integer coordinate pairs in Cartesian space. In other words, if we place each amino acid at a location on a grid with coordinates (x, y) where x and y are both integers, such that each adjacent amino acid in the sequence is also adjacent on the grid, we have "folded" the protein on a 2D lattice. The only additional requirement is that the amino acid walk must be self-avoiding; in this case, we must never visit the same coordinate pair twice in a given fold.

We measure the quality of such a fold by the number of hydrophobic contacts it contains. A hydrophobic contact occurs when a hydrophobic amino acid is adjacent to another hydrophobic amino acid on the lattice, but not in the original sequence. As a result, we are hoping to maximize the number of such contacts. A (loose) upper bound for the number of contacts is 2H, where H is the number of hydrophobic residues - in the best case, each hydrophobic residue will be neighbors with two of its adjacent residues in the sequence, as

well as two other hydrophobic residues (unconnected). We'll measure the quality of our alignment algorithm by the percentage it achieves of a slightly tighter optimum we'll derive below.

With the formalism out of the way, let's dive into the first algorithm you will implement for this project.

## Algorithm Roadmap

The Hart-Istrail algorithm seeks to make some guarantee about the number of hydrophobic contacts a folded protein will have. The algorithm proceeds in a few general steps (do not just code off of this, this is a high level roadmap!):

1. Convert a given amino acid sequence into a hydrophobic-hydrophilic sequence
2. Take the resulting sequence of H's and P's and determine which hydrophobics are at odd positions, and which are at even positions
3. Select a balancing point and odd/even half assignment for the sequence such that the minimum of the odd hydrophobics on the odd side and even hydrophobics on the even side is maximized
4. Suppose, without loss of generality, that the first half of the sequence is the "odd" half (just flip odd and even in the following steps to recover the symmetric case). Then, break the sequence into five segments: the start (all residues until the first odd), the first block (all residues from the end of start to the last odd before the balance point), the middle (all residues from the end of the first block to the first even after the balance point), the second block (all resides from the end of the middle to the last even hydrophobic), and the end (all residues after the last even hydrophobic).
5. Compute "superblock" representations for the first and second block (to be described below).
6. Connect the two superblocks with the start, middle, and end segments

Note that a superblock (e.g. odd, for this description) in the above algorithm places the first odd hydrophobic on a central axis of sorts, and then folds the next odd number of residues until the next odd hydrophobic so that they extend away from this axis, and then back towards it. The following image of an example from the original Hart-Istrail paper might help make this a bit more sensible:

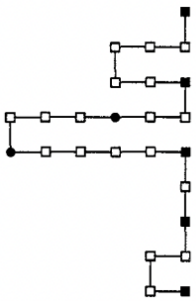


Figure 1: A superblock, as per the Hart-Istrail nomenclature.

This image is a good template for how you should construct your superblocks later.

As you may be able to tell, the final step will have a number of edge cases to be discussed in the algorithm description as well as the next section, which goes into our suggested way of storing and computing a given fold. Note that there's no way of storing lattice positions other than what we suggest below to make this algorithm really clean (that we know of) – if we were to compute the fold explicitly on the lattice, we would deal with many of the same edge cases in a much more convoluted way.

# Fold Representation

In order to represent our final folds of a sequence $s$ of length $n$, rather than place each sequential amino acid on a point on a Cartesian plane (and solving for the fold with such a data structure in mind), we will instead produce a sequence of length $n-1$ consisting of L's (left), F's (forward), and R's (right). We interpret an LFR representation like this as a way of locally "walking" through a fold. For the visualization code we give you with this project, we draw out a fold $f$ of $s$ on a Cartesian grid sequentially starting from $s_1$ at $(0,0)$, where we assume the starting direction of the fold is up, i.e. in the direction of the vector $d =< 0, 1 >$. For each subsequent amino acid $s_i$, we place the associated symbol at the position of $s_{i-1}$ plus whatever we have stored in the direction vector $d$. Then, we augment the direction vector according to L, F, or R stored in $f_{i-1}$ – if L, we change the direction to go left from its current direction, if F, we do not change the direction, and if R, we change the direction to go right from its current direction. Following such a loop will produce a list of positions on the plane, which will be what we consider the associated fold of an LFR sequence.

Less formally, an LFR fold tells us which direction to turn to place the next amino acid at each point in the sequence. Since we start with the direction up, we do not need to modify our first direction, such that we only need $n-1$ LFR symbols to define a full fold. We enforce this constraint in our visualization helper. From here, we'll dive into the algorithm as it relates to this representation.

# Algorithm Details

## Convert the Amino Acid Sequence

We have provided you all a map in `stencil.py`, `HP_MAP`, which maps amino acids to a symbol in $\{\mathrm{H}, \mathrm{P}\}$, where H represents a hydrophobic reside and P represents a hydrophilic residue. Apply this map to the sequences we give you as a first step, as we will do all of our folding in the 2D HP model.

## Determine Odd/Even Hydrophobics

Parse your new sequence in $\{\mathrm{H}, \mathrm{P}\}^+$ into another sequence in something like $\{\mathrm{O}, \mathrm{E}, \mathrm{P}\}^+$. In particular, generate a new sequence that differentiates hydrophobics in odd positions (index is 1 mod 2) from hydrophobics in even positions (index is 0 mod 2). Since we are going for hydrophobic contacts, you don't need to worry about modifying our hydrophilic symbols. This might seem like a weird step for now, but you will explore why we make this distinction in the conceptual questions.

## Select a Balance Point

This is where the algorithm gets a bit more complex. As you will prove later, odd hydrophobics can only contact even hydrophobics (and vice versa). As a result, the goal will be to maximally segregate odds and evens at some position in the middle of our sequence (in order to create two complementary superblocks, as we will examine soon). In this step, you will seek to find this optimal position.

We found success doing this by brute force, i.e. iterating over all $n-1$ possible balance points and checking whether having the odds on the left side of this balance point and evens on the right (or vice versa) maximize the objective function below:

$$\min(\text{odds on odd side}, \text{evens on even side})$$

Let's clarify this general loop structure a bit more. We have $n-1$ possible balance points, given that a balance point segregates our sequence into two non-empty halves. As a result, there is a balance point "between" each adjacent pair of amino acids in the sequence (and there are $n-1$ adjacent pairs in a sequence of length $n$).

Additionally, at each balance point, we can designate the "left" half generated as the odd half or as the even half. For example, if we have 5 odds left of the split and 3 right of the split and 3 evens left of the split

and 2 evens right of the split, the best objective function output we can achieve at this balance point is $max(\min(5,2),\min(3,3)) = 3$. We can easily check both possibilities at each balance point in a nested loop; one will always be worse or equal than the other in terms of our objective function, but we won't always know which side assignment will be better.

As a more complete example, consider the sequence PEPPOPOPPEP. If we select a balance point that segregates this sequence into PEPPOPO PPEP, then we can either assign PEPPOPO to be the "odd" side, such that our objective function value is $\min(2,1) = 1$, or we can assign PPEP to be the "odd" side, such that our objective function value is $\min(0,1) = 0$. If $\min(2,1) = 1$ were the best objective score we had so far, we would store this result, as well as the balance point and side designation.

We add one extra condition, which we will ask you to explain later on. If, at a given balance point, we have an odd (resp. even) hydrophobic adjacent to an even (resp. odd) hydrophobic about the balance point, we skip the case with odd (resp. even) on the left, even (resp. odd) on the right at that point. For example, if our sequence is PEPEOPP, we skip the consideration of the balance point that would divide the sequence into PEPE | OPP, and have odds as right, evens as left. Evaluating this extra condition in this way will save you the headache of some edge cases, while it may (technically) give you one less contact in extreme circumstances that we will not test you on.

After implementing this search, you will have found a balance point that maximizes the objective function we described above. In other words, you will have implemented "argmax" for the objective function we described above, which will output a balance point and a side designation (whether the left side is odd or the right side is odd, or something like that).

As an aside, this objective function is a lower bound for the number of contacts you can generate from a given split. As you will see below, you can fold each side into superblocks such that you achieve at least the number of contacts indicated by this objective function.

## Break the Sequence into Segments

In this step, you will use the balance point you found in the previous step to break up our sequence into useful chunks. Let C = O if your optimal balance point had odds on the left, and E otherwise. Let ¬C be the opposite of whatever C is, i.e. O if the optimal balance point had odds on the right and E otherwise.

The first of the five chunks you will break the sequence into is the start, which we define as the sequence up until you find the first C.

The second chunk goes from the end of the first chunk up until the last C before the balance point you found.

The third chunk goes from the end of the second chunk up until the first ¬C after the balance point you found.

The fourth chunk goes from the end of the third chunk up until the final ¬C of the sequence.

The fifth and final chunk is just the rest of the sequence!

Having this segregation of the sequence will make the next step much easier, as you will see in the connecting superblocks step.

## Form Superblocks

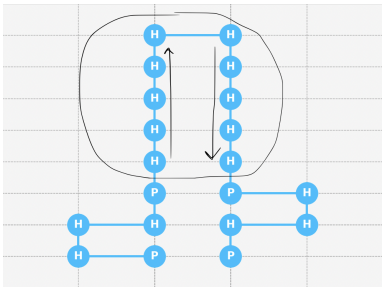
At this point, you will form superblocks out of the second and fourth chunks found in the section above (of length $n_2$ and $n_4$ respectively). In particular, you need to find a way to come up with an LFR sequence of length $n_2 - 1$ to fold the second chunk into a superblock and an LFR sequence of length $n_4 - 1$ to fold the fourth chunk into a superblock.

To fold a chunk with relevant character C into a superblock, start by finding the odd number of residues between each C in the chunk (as a list). Use this information to generate a sequence of LFR moves for each loop in the superblock, as in the example up above (in a loop, over the list of separating residues you just

computed). You should have two cases in this loop: one for if there is only one residue separating C's, and one for if there are three or more residues separating C's.

## Connecting Superblocks

Finally, you need to connect your two superblocks. Connecting them at the middle requires making a small loop using chunk 3 from the last C in chunk 2 to the first ¬C in chunk 4. See the following diagram of one of our sample outputs to get a sense:



Given that this loop will come into the fourth chunk from the top down (direction opposite to our starting direction of $< 0, 1 >$), you likely will not need to "invert" your second superblock in any way.

The last step is to tack on the two ends (chunks one and five) to the two superblocks. As long as these do not intersect your existing structure, you will get full credit on the autograder. Ideally, your start and end will travel down vertically from the start of the second chunk and end of the fourth chunk, respectively.

## Edge Cases

At some point in this algorithm, you may (implementation dependent) need to special case the first or last character of one of these chunks. For example, consider the following two example folds:

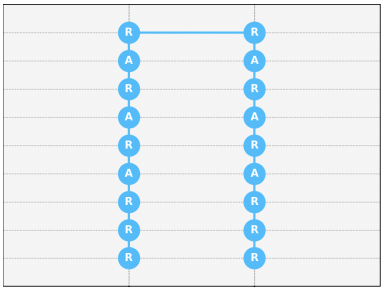


Figure 2: A fold where all residue separations within each superblock are of size 1.

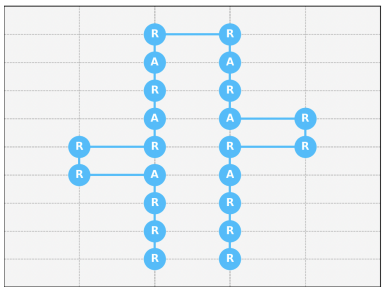


Figure 3: A fold where some residue separations within each superblock are of size 1, but not those bordering the start and end of the sequence.

Given both sequences are symmetric, both the forward and backward versions of these folds, in LFR representation, will be accepted by the autograder. However, for the sake of the edge case argument, suppose we started from the rightmost R on the bottom of this fold (instead of the leftmost). In this case, when you exit the second superblock, which ends at the A four residues up from the bottom left, you need to go forward in the top photo, versus right in the bottom photo (more generally, you need to go forward if the last set of separating residues in block 2 is of size one, and right otherwise). You'll notice, though, that if you program this carefully from the beginning and follow the steps we've outlined, that you can avoid such edge cases; i.e., do you need to worry about this type of edge case at the start/block one juncture, block one/middle juncture, middle/block two juncture, or block two/end juncture if you start from the leftmost residue instead? Something to think about.

Don't overthink this section – if you follow the roadmap we've written closely, you very well might never run into this type of edge case. However, given we are using LFR representation, we expect bugs associated with this problem to be fairly common. You'll know you might be up against this situation if your program is working well, but when you try it with only one separating residue in the first or last sub-block of either superblock, one of your superblocks gets inexplicably rotated ninety degrees.

## Visualization

We have provided visualization code in `stencil.py`, which you can use at your discretion for visualizing each fold your algorithm produces. You can also run the shell script `visualize.sh` on a sequence and a fold to visualize your results from the terminal. Make sure to use this when requested in the following sections! You can modify `visualize.sh` and `stencil.py` as much as you would like, given that we will not be autograding these files in any way. Note that this code requires the installation of `matplotlib`, which we are happy to help with in office hours if it is of any concern.

# Part One (40 Points)

Now that we've looked at the specifics of the algorithm (without much intuition for what it's doing), let's see if we can understand why it works, and what it can achieve in terms of number of hydrophobic contacts. Answer the following questions in `theory.pdf`:

1. Explain why an odd hydrophobic (resp. even) can only make contacts with an even hydrophobic (resp. odd). (10 points)
2. As mentioned earlier on, a very loose upper bound for the number of contacts is two times the number of hydrophobics in the sequence. Since odds cannot contact odds and evens cannot contact evens, we can restrict this upper bound to $2 * \min(\mathcal{O}, \mathcal{E}) + 2$. Why can we make this restriction? Explain. (6 points)
3. Give an example of an HP sequence and LFR fold that actually achieves this upper bound. Feel free to draw out the fold for us, rather than actually writing out your LFR sequence. (4 points)
4. Now that we have an upper bound in mind, let's prove the effectiveness of our algorithm asymptotically (i.e., ignore edge cases and consider the ephemeric optimum derived in 2 to be $2 * \min(\mathcal{O}, \mathcal{E})$). Assume without loss of generality that $2 * \min(\mathcal{O}, \mathcal{E}) = 2 * \mathcal{O}$ in our following proof, i.e. that the odds are the limiting factor. We could easily invert the following proof for the other case. Also assume that both $\mathcal{O}$ and $\mathcal{E}$ are even for simplicity – the proof can be extended to odd with a few caveats, but with some careful floor/ceiling notation we don't want to put you through.
   a) Prove that there exists a balance point such that there are exactly $\frac{\mathcal{O}}{2}$ odd hydrophobics on one side of the balance point and $\geq \frac{\mathcal{E}}{2}$ on the other. (5 points)
   b) Prove that the Hart-Istrail algorithm can use such a balance point to construct a fold that has at least $\frac{\mathcal{O}}{2}$ hydrophobic contacts. Conclude from our upper bound that the asymptotic approximation ratio for the Hart-Istrail algorithm is $\frac{1}{4}$. (5 points)
5. Explain why we added the case in our balance point search to avoid balance points that directly segregate on a hydrophobic-hydrophobic connection and include both hydrophobics about the balance point in the respective superblocks (i.e., have odds as left and evens as right in an O|E split or odds as right and evens as left in an E|O split). What happens to the middle chunk we described in the sequence segmentation step of the algorithm if we split on one of these cases? Why is this a problem? (4 points)
6. Include screenshots that visualize your algorithms outputs on the following three sequences. Please visualize these sequences in their **HP form**. (6 points)
   a) `RYSHMYGSSKMVNPHKYISHNK`
   b) `ILGQSTLFPPWVMVIG`
   c) `WHHEEKLSGRRCNNTDLKCPRVTESRRFNKLKEFRSARQF`

## Part Two (60 Points)

Now that we have a detailed understanding of the algorithm and what it guarantees, it's time to implement it! Our autograder requires that your program follows the following argument specifications:

```
$ sh istrail.sh <seq.txt>
```

And outputs a fold as a sequence of symbols in the set $\{L, F, R\}$ as described above. We will test whether your folding algorithm works by checking if it gets at least as many contacts as stipulated by the maximum objective function value described in the balance point part of the algorithm. See the following examples:

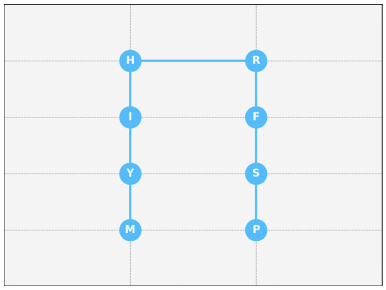
`seq.txt`

```
MYIHRFSP
```

Output:

```
> sh istrail.sh seq.txt
FFFRRFF
```

Visual Output:



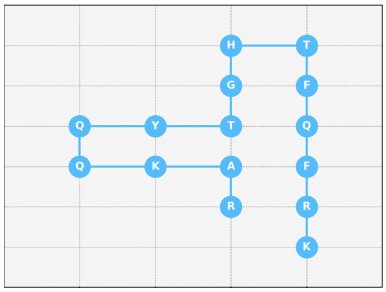

`seq.txt`

```
RAKQQYTGHTFQFRK
```

Output:

```
> sh istrail.sh seq.txt
FLFRRFLFRRFFFF
```

Visual Output:




Note that while your folds should achieve the contact lower bound, have a roughly similar structure to ours, and have the same balance point (as defined by chunk separation) on these examples, your LFR sequence does not need to exactly match what we have.

You will be tested on a number of public edge cases, although we will refrain from testing you in cases where it is not possible to make any contacts. As a result, you don't need to check for such cases.