SBI-Python project

Carla Castignani, Alexander Gmeiner, Nerea Moreno.

Initial Problem

The main aim of this project is to build a protein complex with a set of interacting pairs. Using the information of the protein-protein interactions, we should be able to return a CIF with the whole structure.

Theoretical Background

Introduction

Protein macro complexes are involved in many cellular functions and thus, their characterization is essential to understand many several biological processes. In this way, the multi-scale modeling of macro-complexes, such as viruses, nucleosome or ribosome is crucial in the understanding of this processes.

In the last years there has been a large influx of new annotated structures of protein complexes due to advances in x-ray crystallization and visualization techniques. However, a vast numbers of interacting complexes are still not annotated by experimentally determined 3D structures [1]. This is due to nature of the protein complexes - they could be as simple as a stable symmetric homodimer or as complicated as a multi-component heterooligomer. Thus, stabilizing these complexes for X-ray crystallization is a very challenging task.

Methodology

So as to solve the mentioned problem, we have designed a program able to deal with the interacting pairs and later on built a model from it. The program can be used in different ways, but it always starts with a first superimposition of the files of the current folder. If this files are enough to build the macro complex, no other functions are need to be used. However, the user can also provide only the essential interacting pairs of the macro complex and then, the program will repeat this subunits iteratively as many times as wished.

As for the first part of the program, in order to reconstruct a protein complex from several pairs of chains, we need to know which chains are the same so we can superimpose them. We have approached this using the names of the files. Thus, the program first reads the files and creates a list of pairs to superimpose based on the file names. After the identification of chains, the following step is to locate the components of our complexes into 3D space. This is done by means of the superimposition, using the module Superimposer from Biopython.

Steric clashes are a major tool in protein modeling, complex building and assessment of ab initio model building. They occur when nonbonding atoms of the model overlap in a non favorable way [2]. Thus, after having add each new chain, we search for contacts and delete the chain if it has clashes.

With this steps, our program is able to build a model from the PDB files chosen of the current folder however, sometimes this could lead to non-meaningful biological structures. For example, in cases where the user only provides the essential interacting pairs. In order to overcome such challenge, we implemented another function able to build a macrocomplex from a initial subunit iteratively. This function is only performed if the user specifies it in the arguments when running the program.

This second part of the program takes as input the initial built PDB and understands it as a basic subunit of the final macro complex. In other words, the structure that we need to repeat n number of times in order to obtain the final model. Again, knowing which chains are similar remains a key step in the process. In this part of the program we approach this by using pairwise sequence alignment (using the Pairwise2 module from Biopython). BIO.pairwise2 uses a dynamic programming algorithm to perform pairwise sequence alignments. A local alignment finds just the subsequences that align the best. In this way, we only superimpose structures that have at least two identical subunits (pairwise sequence identity > 95%). The score we consider to decide if two sequences are or not superposable is based on the identical positions divided by the length of the alignment.

The user has the option to decide the number of iterations. So, after deciding which pairs of chains from the subunit have enough identity to try and be superimposed, the program tries to superimpose each one of them as many times as required by the user, or until finding clashes for the chains that need to be added to the structure.

The program will choose one pair and try to perform a simple superimposition, if the superimposition does not work because there were clashes between the new chain and the existing structure, it will try with another pair, and the previous one is removed from the list of pairs to superpose, in order to make sure of not checking that path twice.

If the superimposition does work, then the structure is saved (a deepcopy needs to be performed as we are iterating on it) and the program tries to superimpose the subunit again in the amplified structure following the same match. This means that if in the first iteration, A-B were superposed successfully, the chain that is added to the structure is renamed (because new chains need to have names that are different from those already existing in the structure), and we actualize the match list with the name of the new pair. Whenever a superimposition in a given path fails, the pair is removed from the list and the next path is checked until removing all the possible pairs from the list. Every time a path arrives to the final number of iterations asked by the user successfully, the structure is saved and will be written into a CIF format file once the program finishes running.

When the program finishes calculating the possible structures given a number of iterations, it will iterate in the list of structures that has been built, and it will create a CIF file for every structure built.

If the program is not able to build any structure with the given number of iterations, then it is advisable to try with a lower number of iterations.

Installation

1. Download the zip file BOB_the_builder.tar.gz
2. Unpack using

```
$ tar -zxvf BOB_the_builder.tar.gz
```

3. Change directory to BOB_the_builder

```
$ cd BOB_the_builder/
```

4. Two possibilites:
    - Copy

    copy BOB_the_builder.py, macrocomplex.py and superimposer.py into your directory with the input files and run with desired flags (see description)

    ```
    $ python3 BOB_the_builder.py
    ```

5. - Install packages
6. - Unpack BOBthebuilder_pkg-1.0.tar.gz

    ```
    $ tar -zxvf BOBthebuilder_pkg-1.0.tar.gz
    ```

    - Change into the new directory BOBthebuilder_pkg-1.0

    ```
    $ cd BOBthebuilder_pkg-1.0
    ```

    - Install the packages using PIP3

    ```
    $ pip3 install .
    ```

    - Copy BOB_the_builder.py into your directory with the input files and run with desired flags (see description)

    ```
    $ python3 BOB_the_builder.py
    ```

General Information

- Prerequisites:

This program works on Python 3.0 release (https://www.python.org/download/releases/3.0/). Required modules: argparse, os, copy, sys and BIO.PDB.

- Input files

The input files of the program must consist on PDB files with the interacting pairs. Some considerations need to be taken into account:

1. The PDB file names must have the names of the chains of the interacting pairs before the .pdb extension. Example: PAIR_XA.pdb, XA.pdb. Wrong input filename would lead to mistakes in the macrocomplex builder. Example: X_A.pdb
2. The order of the chains must be consistent between the filename and the PDB file. Example: If filename is XA.pdb, in the PDB file chain X will appear first.
3. Chain names inside the PDB file don't have to be consistent with the filename. The program will produce a correct output as long as the order of the chains is preserved. Example: Filename is XA.pdb, but inside PDB they are chain A and chain B.

Tutorial

Command line options

The arguments considered to fill the ArgumentParser object can be achieved by:

```
$ python project.py -h (or --help)
```

- -o --out_file: Output file name - Without extension. Required.
- -c --chain_names: Chains with which you want to build the macrocomplex.
- -i --iterations: Number of times you want to repeat the basic subunit.
- -coff '--cut-off: Specify the minimum cut-off distance for clashes search in angstroms. 1.1 by default.

If the program is ran without any other argument than the output filename, It will build the macrocomplex with all the PDB files located in the current directory. It will not take into account repetition of each subunit in the final macrocomplex. Thus, It can lead to non biologically meaningful structures, the user must then evaluate the final output file.

If the program is ran with the chain_names option, it will only use the chains selected by the user to create the macro complex. Beware that, if the names of the chains do not correspond to the file names or there are no files for those chains, the will produce wrong outputs.

The program can also be ran using the iteration option. The iteration option is used to form multimeric proteins. The number of iteration corresponds to the number of times the complex contains the basic subunit. The components of the basic subunit are defined by the user with the --chain_names options. The subunit is constructed with the given interaction pairs and will then be further added to build the complex structure.

Modules

- **superimposer.py:**
    - **clash_search(ref_structure, new_chain, cut_off):**

      This function searches for the neighbor atoms within a certain distance of a reference atom. It will go through the structure chain by chain and check if the new chain we want to add finds neighboring atoms within the given cut-off.

Input:

- ref_structure: the reference structure that the sample structure will be added to
- new_chain: Name of the new chain to be used as reference for the neighbor search
- cut_off: the minimum distance between the atoms of the new_chain and the existing structure that will allow the chain to be added

Output:

- if it finds neighbours within the cut-off it returns: TRUE
- if not it returns: FALSE

- **superimposition(struc_ref,struc_sample,chain_ref,chain_sample,current_chain1, current_chain2,chain_dict, cut_off):**

  This function calculates and applies a superposition matrix of the sample structure onto the reference structure. Further it returns a new structure with the sample structure added to the reference structure.

  Input:

  - struc_ref: the reference structure that the sample structure will be added to
  - struc_sample: will be the sample structure in the superimposition process and will be added to the reference structure
  - chain_ref: chain from the reference structure to be superimposed
  - chain_sample: chain from the sample structure to be superimposed
  - current_chain1/2: contain the integer that will be converted to a character and used to add new chains to the structure
  - chain_dict: dictionary containing the original name of a chain in the structure as key and all the new names it acquires as values
  - cut_off: the minimum distance between the atoms of a new chain and the existing structure that will allow the chain to be added

  Output:

  - NEW_STRUCTURE (with sample chains added) and the updated chain_dict dictionary

- **macrocomplex.py:**
  - **matcher(structure_cif):**

    This function takes a file in CIF format and returns a list with the pairs of chains sharing >95% of identity and the structure

    - Input: filename of a file in cif format containing the structure that we want to use to build the macrocomple
    - Output:
      - match_list: a list of lists containing the pairs that share enough homology to be superimposed
      - subunit: the structure parsed from the input file
  - **trial(new_structure,n,current_chain1,current_chain2,match_list,start,cut_off):**

    This function takes two structures, a number of iterations and a match_list (among others) and returns "False" or a macrocomplex.

- Input:
    - new_structure: this will be the reference structure and is the original subunit
    - start: will be the sample structure in the superimposition process and it is the original subunit
    - n: number of iterations given by the user to build the macrocomplex. This is the number of subunits that form the complex -1.
    - current_chain1/2: contain the integer that will be converted to a character and used to add new chains to the structure
    - match_list: contains the list of pairs of chains that could be superimposed
    - cut_off: the minimum distance between the atoms of a new chain and the existing structure that will allow the chain to be added

  A new trial is started while the match_list still contains pairs of chains that could be superimposed. A trial ends when an attempt to superimpose two chains fail, those two chains need to be removed from the match_list, to not be tried again. A trial also ends when the number of superimpositions reaches the number of iterations indicated by the user. Also in this case the matc_list is actualized.

  - Output:
    - FALSE will be returned when more superimpositions cannot be performed with a given pair of chains and the number of iterations was not reached
    - NEW_STRUCTURE will be returned if a pair of chains reaches the number of superimpositions required

- **jump(new_structure,original,j,current_chain1,current_chain2,skip_chain,chain_dict,n,match_list,cut_off):**

  This function takes two structures, a match_list (among others) and returns False or a new structure (among others).

  Input:

  - new_structure: the reference structure (in the first iteration it is the original one, in further iterations it is the amplified one)
  - original: will be the sample structure in the superimposition process and it is the original subunit
  - n: number of iterations given by the user to build the macrocomplex. This is the number of subunits that form the complex -1.
  - current_chain1/2: contain the integer that will be converted to a character and used to add new chains to the structure
  - match_list: contains the list of pairs of chains that could be superimposed
  - cut_off: the minimum distance between the atoms of a new chain and the existing structure that will allow the chain to be added
  - skip_chain: in the first iteration it is set to the reference chain in the superimposition, then it is maintained
  - chain_dict: dictionary containing the original name of a chain in the structure as key and all the new names it acquires as values
  - j: number of the current iteration

Every JUMP is an attempt to superpose two chains that are in the first item of the match_list. When those chain matching the list are found the superimposition is tried and from its return value this function will perform different operations. If the return is False, meaning that the superimposition didn't work, the match is removed from the list and False is returned. If the return value is not false, then the new structure is returned and the match list is actualized to the new names of the matching chains

Output:

- FALSE will be returned when more superimpositions cannot be performed with a given pair of chains and the number of iterations was not reached
- NEW_STRUCTURE will be returned if a pair of chains reaches the number of superimpositions required

Both FALSE and the NEW_STRUCTURE are returned with other values that the functions need to keep track of

- **superimpose(ref_structure,original,n,ref_chain,sample_chain,skip,current_chain1,current_chain2,chain_dict,j,cut_off):**

  This function takes reference and sample structures and chains (among others) and returns false or a new structure (among others).

  Input:

  - ref_structure: the reference structure (original subunit or amplified one depending on the iteration number)
  - original: will be the sample structure in the superimposition process and it is the original subunit
  - n: number of iterations given by the user to build the macrocomplex. This is the number of subunits that form the complex -1.
  - ref_chain: chain from the reference structure to be superimposed
  - sample_chain: chain from the sample structure to be superimposed
  - skip: chain that needs to be skipped when adding chains to the structure with transformed atoms (in the first iteration it is a chain belonging to the original subunit)
  - current_chain1/2: contain the integer that will be converted to a character and used to add new chains to the structure
  - match_list: contains the list of pairs of chains that could be superimposed
  - chain_dict: dictionary containing the original name of a chain in the structure as key and all the new names it acquires as values
  - j: number of the current iteration
  - cut_off: the minimum distance between the atoms of a new chain and the existing structure that will allow the chain to be added

  The superimpose tries to superpose the chains that are given by the jump function, it will try to add the chains from the sample structure to the one that is being amplified. Then it will check for clashes calling a clash_check function. If the return value is True, it detaches the new chain that had been added to the structure and returns False and other variables to the JUMP function. If the return value is False for all the chains that need to be added, then the function will return a deepcopy of the structure amplified among other variables.

Output:

- FALSE will be returned when more superimpositions cannot be performed with a given pair of chains and the number of iterations was not reached
- NEW_STRUCTURE will be returned if a pair of chains reaches the number of superimpositions required

Both FALSE and the NEW_STRUCTURE are returned with other values that the functions need to keep track of.

Analysis of examples

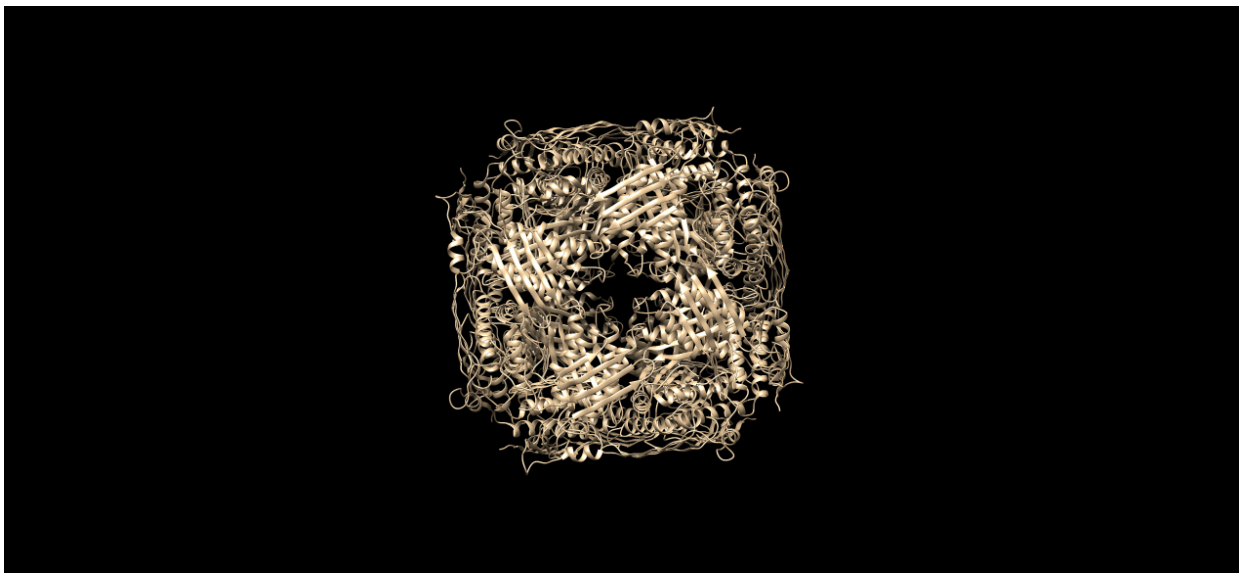
Following we will see the analysis and execution of some example structures in order to provide a better understanding of our program.

1. **Imidazoleglycerol-phosphate dehydratase (6EZM)**

In this example, we have a set of pairs organized in multiple files of interactions between chain X and all the other chains of the complex. In this case then, all the information contained in the files is enough to directly build the complex. Thus, to achieve this complex we can run:

```
$ python3 BOB_the_builder.py  -o output
```

With this options we are setting the name of the output file to “output” and taking all the files of the current folder as the ones necessary to build the structure. We get then a structure with no clashes that looks like this (running time *4sec*) :

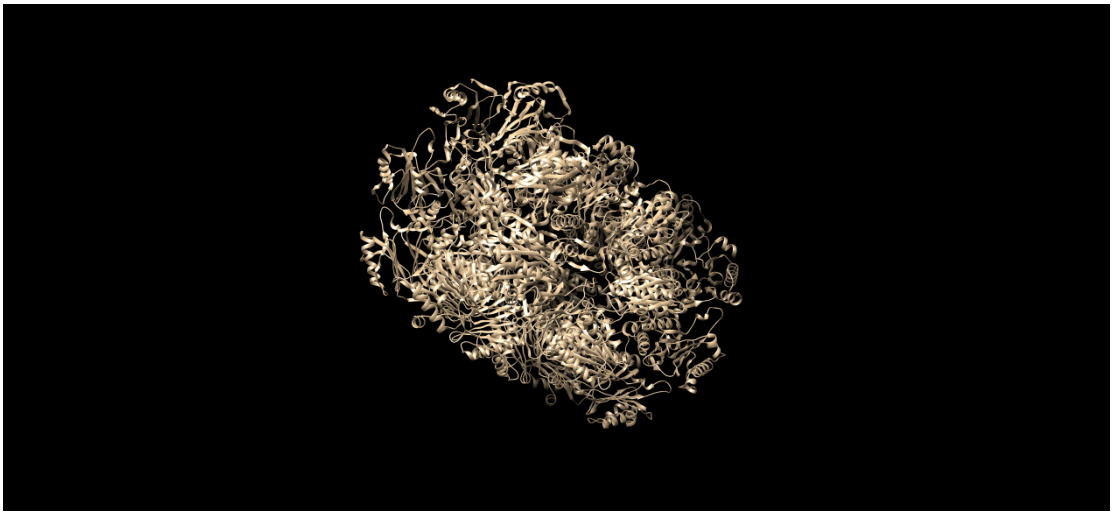


2. **Proteosome**
    - Example 1 (1G65):

      In this case we have a folder containing some interacting pairs, some of them are essential to build the macrocomplex while others are not contained in the complex. Thus, using the -c option we choose to build the model with only those chains we know are necessary to build the complex. Furthermore, using the -i option we can decide how many times this chains should be repeated in the final structure. Thus, to achieve this complex we can run:

```
$ python3 BOB_the_builder.py  -o output -c EFS2O -i 7
```

Trough our sequence similarity analysis based on a pairwise alignment our program can identify which chains are similar and use this information to build the model. After checking the contacts, we obtain an structure that looks like this (running time *52sec*) :

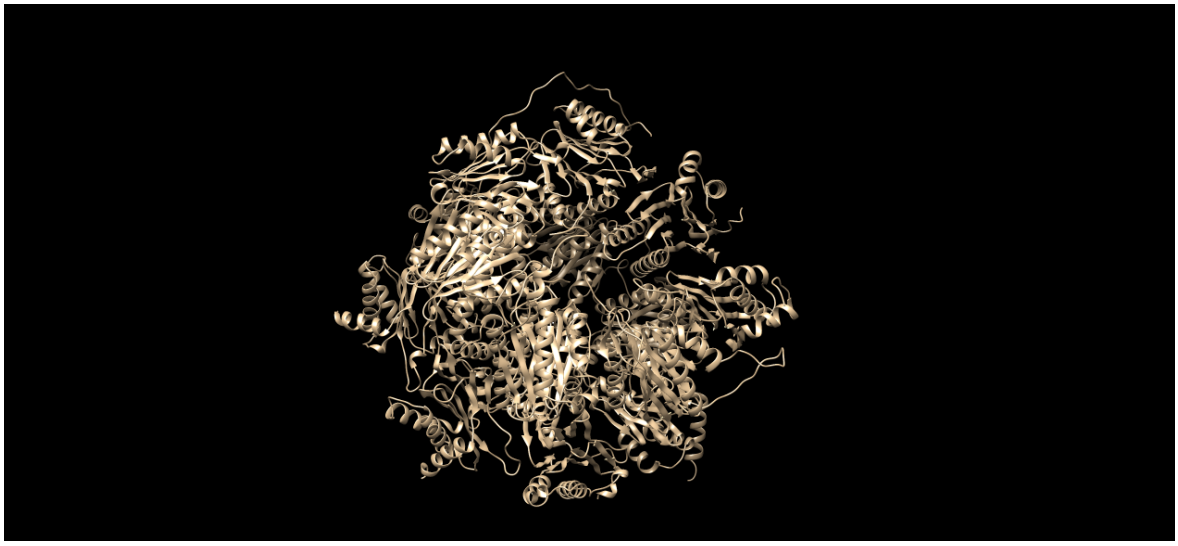


- Example 2 (3H4P):

  This example corresponds to another 20S proteasome, this time the core. In this case, we have all the interacting pairs. Thus, we can run the program as following:

  ```
  $ python3 BOB_the_builder.py  -o output
  ```

  We then get a structure with no clashes that looks like this (running time *4sec*):

When we compare both examples, we notice that in the cases where the user gives to the program only some protein-protein interactions, this suposes a higher computational than running the program with all the pairs. This is mainly due to the fact that in the first example, the program has to identify the chains using sequence similarity analysis. Besides, as explained before, in this option, we need to copy our structure in every iteration so as not to overwrite it.

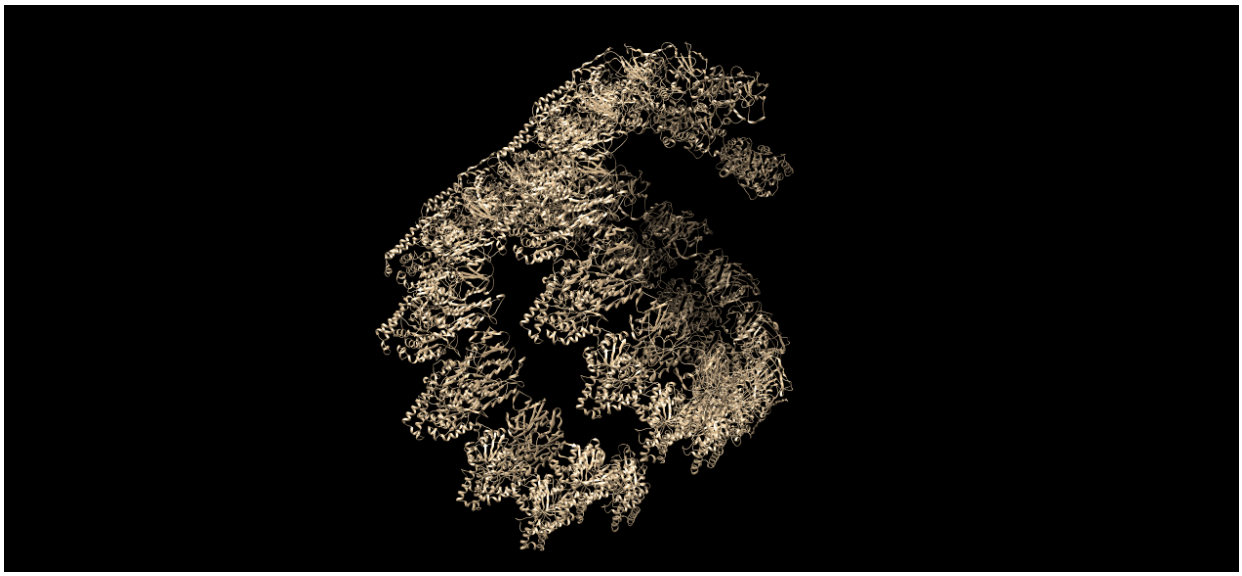
This results in a computational cost that grows as the structure is built.

3. Microtubules (**1TUB**)

In this case we have two interacting pairs, consisting of the tubulin alpha-beta dimers. In this case, we have selected the desired chains to build the model and chosen 21 iterations. This is an arbitrary number, as we know that this structure could go on forever without ever having contacts without the chains. Additionally, our program has a maximum number of iterations permited (100), as we considered that structures with high number of iterations could lead to very large files. In order to build the model, we run the program as followed:

```
$ python3 BOB_the_builder.py -o output -c xXZW -i 21
```

In this case, the sequence similarity analysis used in our program to identify the similar chains to build the model is not taking long, as there are only two files. However, the structure we are building is considerably larger than other and thus, the copy takes longer. After checking the contacts, we obtain an structure that looks like this (running time 7min 21sec) :



Limitations

1. Dependency on the file names: Our program is intimately dependent on the names of the PDB files to construct the model, as it uses this parameter to identify similar chains.
2. Scalability: In order to build the complex in the iterations options, we need to perform a copy (deepcopy) of the structure. This proceeding has a computational cost that grows exponentially. Thus, we have noticed that our program has a good benchmark in small structures but this decreases in larger structures.
3. Complex structures: In this project we have noticed the complexity of some macro complexes. Our program doesn't the iterations to continue when more than two chains overlap (clashes found). However, in spherical structures (such as virus or nucleosome), we find ourselves in a situation in which we should be adding those chains, as even though some chains are clashing, others are giving meaningful information.
4. Unable to model protein-RNA or protein-DNA interactions as this option is not

implemented yet on our script.

Bibliography

[1] Soni, N., & Madhusudhan, M. S. (2017). Computational modeling of protein assemblies. *Current opinion in structural biology*, *44*, 179-189.

[2] Ramachandran, S., Kota, P., Ding, F., & Dokholyan, N. V. (2011). Automated minimization of steric clashes in protein structures. *Proteins: Structure, Function, and Bioinformatics*, *79*(1), 261-270.