Reconstruction of a macro-complex using interacting subunits

Lydia Fortea Juan Luis Melero

Contents

Chapter 1

Background

The aim of the project is to reconstruct a marco-complex having only the pair interacting chains, using a standalone program created by ourselves. The program we created is based on several bioinformatic features, including structural superimposition and sequence alignment, among others.

1.1 Protein-Protein Interaction and Complexes

An important point of the project is understanding the Protein-Protein Interaction and Complexes. In the quaternary structure of a protein, there are more than one separated chains of proteins that interacti between them. The interaction of these chains can involve a lot of intermolecular bounds, such as hydrogen bounds, electrostatic interactions, pi stacking, cation-pi interaction, etc. This diversity of interactions makes the protein-protein interaction very common in order to stabilized the molecule and generate a biological function.

The whole structure, where two or more chains are combined and have one or different functions, is called a complex. The formation of a complex can be made by protein-protein interaction only or nucleotides (DNA or RNA) can also be part of a complex if there is DNA-protein or RNA-protein interactions.

Focusing on the project, having the protein-protein interaction by pairs, we want to reconstruct the whole macro-complex.

1.2 Sequence Alignment

Pairwise sequence alignment is done for several reasons. One of them is because we want to compare the chains from the same interacting file to know if it is a homodimer or a heterodimer. Another reason is to compare different interacting files, in order to know if they are the same pair of interacting chains or they are different. The last one, is to know which chains must be superimposed, in case the interacting chains are different heterodimers (see section Modules and Packages, subsection Homodimers and Heterodimers).

1.3 Structural superimposition

We cannot assume that the protein-protein interacting pairs are well oriented in the space. Therefore, in order to give to each part the correct orientation, we do a structural superimposition. Structural superimposition allows us to put the chains in the correct spatial orientation, and we use sequence alignment to know which chains must be superimposed. With superimposition, all the chains will be well positioned.

Chapter 2

Algorithm and Program

2.1 Inputs and Outputs

2.1.1 Input files

The program takes interacting pair chains for building the macro-complex. However, in the command line, the input must be one of the following:

- The list of PDB file names which contains the interacting pair chains.
- A directory that contains all the PDB files of interacting pair chains.
- By default, if no input is indicated, it takes the current directory as input, and takes all the PDB files as interacting pair chains.

Input files must be all in the directory specified in the arguments or in the current directory by default. The program will read all the PDB files, so it is necessary that all the PDB files in the working directory are the subunits of the macrocomplex and nothing else.

2.1.2 Output file

The output file will be one PDB file in which there will be the coordinates of the atoms of the macro-complex. The output file will be located in the directory specified in the arguments. Depending on the inputs (homodimer, heterodimer or mixed), the output will have different names in the current directory (homodimer$_{c}omplex.pdb, heterodimer_{c}omplex.pdborhomodimer_{h}eterodimer_{c}omplex.pdbifmixed).Ifthecur$

2.2 Modules and Packages

Biopython is the principal package used, as well as *sys*, *os*, *re*, *argparse*, *subprocess* and the self-created modules *homodimers* and *heterodimers*.

2.2.1 Biopython

Biopython is the main open-source collection of tools written in Python to work with biological data. From Biopython we take the following subpackages:

- Bio.PDB, to work with PDB files. From Bio.PDB we use the following subpackages:
  - PDBParser, to parse PDB files and obtain Structure Objects to work.
  - CaPPBuilder, to create the sequences taking into account Ca-Ca distance.
  - PDBIO, to save the structures into a PDB file.
  - Superimpose, to execute structural superimposition between structures.
- Bio.pairwise2, to align protein sequences one by one

2.2.2 sys

Sys package is the System-specific parameters and functions. This package is used to read the arguments in the command line (sys.argv) and to have access to the three channels of communication with the computer: the *standard in* (sys.stdin), the *standard out* (sys.stdout) and the *standard error* (sys.stderr).

2.2.3 os

Os package is the Miscellaneous operating system interfaces. With this package, the program can access synonimous commands of the shell, allowing the program to, for example, change working directory (*cd* in shell, *os.chdir* in python). This package is usefull to call command lines from the system but without consuming as much CPU.

2.2.4 re

Re package in the Regular expression operations package. It allows to work with regular expressions with python. In the program, it is used to find PDB and FASTA files, searching the extension of FASTA (.fa, .fasta) and PDB (.pdb) as regular expression at the end of the files.

2.2.5 argparse

Argparse package is the Parser for command-line options, arguments annd subcommands. This packages allows to include the options for the user. The options included in the program are described in section *Options and Arguments*.

2.2.6 subprocess

Subprocess package is the subprocess manager that allows us to use bash and shell commands. In our program is used to call Chimera Software if the option -vz, --visualize (see section Options and Arguments) is active.

2.2.7 Homodimers, Heterodimers and Common Functions

Homodimers and Heterodimers are self-created modules which analyze the structures depending on the input files. If the pair of chains in the interacting files are the same (A-A), then it is considered homodimer. If the pair of chains in the interacting files are different (A-B) but all the files contain this heterodimer (files are A-B, A-B...), then it is considered *repeated heterodimer*. Finally, if the pair of chains in the interacting files are different and all interacting files are different (A-B, B-C...), then it is considered *heterodimer*. For homodimers and repeated heterodimers, homodimers module is used. For heterodimers, heterodimers module is used.

Homodimers module

Homodimers module (*homodimers.py*) contains two functions:

- get_structure_homodimer, that is used when the input files are all homodimers
- get_structure_mix, that is used when the input files are homodimers and heterodimers.

Both functions are explained in section Functions.

Heterodimers module

Heterodimers module (*heterodimers.py*) contains two function:

- align_sequences_heterodimers
- superimpose_structures_heterodimers

Both functions are explained in section Functions.

Common Functions Module

Common functions module (*common_functions.py*) is a module that contains general functions that will be used during the program. This functions are:

- get_sequence
- seq_comparison
- chains_comparison
- temp_structure
- clash_identifier
- save_complex

All these functions are explained in section Functions.

2.3 Functions

In the program there are a lot of functions that will be used during the process of building the macro-complex. These functions are explained below.

2.3.1 get_input

This function handles the input argument. It uses regular expressions to find those files ended with ".pdb" and puts them into a list. If the input is a list of files, then it puts them directly into a list. The function returns a list that is the PDB interacting files that will be used to build the complex.

2.3.2 get_name_structure

This function uses regular expressions to return the filename of the PDB file.

2.3.3 common_functions.get_sequence

This function deletes heteroatoms and returns the sequence of the proteins with more than 40 aminoacids (to avoid ligands). It uses CaPPBuilder package.

2.3.4 common_functions.seq_comparison

This function is used to find identical sequences. It performs a pairwise sequence alignment and if the percentage of identity (%id) is greater than 0.95 it returns True, else, it returns False. It uses Bio.pairwise2 package to make the alignments.

2.3.5 common_functions.chains_comparison

This function compares the chains of the interacting pairs and returns Frue if they are the same or False if they are different. It uses seq_comparison function.

2.3.6 get_pdb_info

This function analyze all the PDB files from the input. It extracts and compare the structures and their sequences in order to know if we have heterodimers o homodimers. It returns the pairwise interactions, the homodimer interactions and the heterodimer interactions.

2.3.7 common_functions.temp_structure

This function creates a temporary structure with the interactions. It returns the built structure.

2.3.8 common_functions.clash_indentifier

This function checks the final structure if there were any clash. It returns True if there are clashes or False if there is not any clash. If there were any clash, the chain is not added.

2.3.9 common_functions.save_complex

This function saves the structure into a PDB file. It return nothing, but creates the PDB output file. This function uses Bio.PDBIO package.

2.3.10 heterodimers.align_sequences_heterodimers

This function extracts all the sequences of the chains and aligns them with pairwise alignment and stores the percentage of identity (%id). Those chains with a %id greater than 99% are assumed to be the same and then, they are going to be superimposed.

2.3.11 heterodimers.superimpose_structures_heterodimers

This function is used to superimpose different structures. The chains that will be superimposed are those with the best alignment from *heterodimer.align_sequences_heterodimers*. We do the superimposition taking one chain as reference chain and the other as moving chain. Then the coordinates are updated and the reference chain is deleted in order not to have repeated superimposed chains. After the superimposition, one of the chains superimposed is removed not to have repeated chains in the structure. It returns the structures superimposed.

2.3.12 homodimers.get_structure_homodimer

This function is called when the input files are homodimers or repeated heterodimers. It superimposes the structures, taking one as reference and correcting the coordinates, and afterwards eliminates one of the superimposed chains not to have redundancies. It returns the built structure.

2.3.13 homodimers.get_structure_mix

This function is called when there is a mix between homodimers and heterodimers in the input files. First, it aligns and superimposes homodimers like function *homodimers.get_structure_homodimer* and after that it adds heterodimers interactions, superimposing them like heterodimers module. It returns the built structure.

2.4 Workflow

First of all, we take the input files provided by the user using *get_input* function. We extract the information from these PDB files to know if we are dealing with homodimers (A-A interactions), repeated heterodimers (A-B, A-B...) or distinct heterodimers (A-B, B-C...). To do that, we use *get_pdb_info* function.

If the interactions are homodimers (A-A), repeated heterodimers (A-B, A-B...) or a mix of homodimers and heterodimers, then we use the standalone module *homodimers.py*. If the interactions are distinct heterodimers (A-B, B-C...), then we use the module *heterodimers.py*. You can see the workflow of each module in section *Modules and Packages*, subsection *Homodimers and Heterodimers*.

Finally, the program creates and saves a PDB file with the final structure using *temp_structure* and *save_complex* functions. If the option visualize (-vz, --visualize) is activated, it opens the PDB output file with chimera (see section *Options and Arguments* for requirements of this option).

2.5 Restrictions and Limitations of the Program

For heterodimers, one of the main restrictions is that all chains must be able to be followed. That is, we must be able to build a path joining all the subunits (A-B, B-C, C-D...). If it is not like that (A-B, C-D... for example), the program will crash. This is because one chain will be used as reference chain, and the other as moving chain. If it only appears once, then it is not possible to build the structure.

There are many problem handling homodimer interactions or heterodimer interaction. The main one is that structural superimposition is not useful to rebuild the protein. Moreover, we do not consider the stoichiometry (how many repetitions there are) and we assume that, the same interaction is only once and, if it were repeated, the user will provide it again. We could ask for the stoichiometry to the user to avoid this problem, and change the code to handle this.

The program does not work for protin-nucleotide interaction. To do this, we should analyse also nucleotide structures in PDB and change the code to work with nucleotides.

The program does not check if the PDB files are well-formatted. If there were any error in the input files, it could rise an error at some point of the program or create an output with the wrong-formatted data that, obiously, will not correspond to the correct macro-complex. If we knew what are the most typical errors or how a PDB cannot be well-formatted, we could rise an error before the script.

The program does not take into account proteins with less than 40 aminoacids. This is made to avoid peptide ligands. Therefore, small proteins will not be considered in the program. We could solve this if we knew the nature of the PDB structure, if ligand or protein.

We only create one model. There program could generate many models and compare them (e.g. analysing the energy) and then choose the best structure. Moreover, there is no evaluation of the quality of the model and the structure created.

If different chains have the same name, they are added as new structures. This is done because Chimera has problems recognizing differet chains with the same name. Therefore, although they could be the same chain, the name is changed.

Chapter 3

How to use the program

3.1 Requirements

The main program requires for *Biopython* package and auxiliary modules *homodimers.py*, *heterodimers.py* and *common_functions.py*, as well as all preinstalled Python packages indicated in *Modules and Packages* section.

3.2 Options and Arguments

In the program the following arguments are available:

- -i, --input; it takes the directory as input. By default, it takes the current directory.
- -o, --ouput; it takes the directory and the filename of the PDB file. By default, it creates a file called *output.pdb* in the current directory.
- -s, --sequence; it takes the directory and the filename of the FASTA file in which there is the sequence of the macro-complex. By default, it takes de value *None* and runs the program with default parameters.
- -v, --verbose; if this option is active, it prints the log to the standard error.
- -vz, --visualize; if this option is active, at the end of the script it opens the output file with Chimera. It will only work if Chimera is installed and the software is in `/usr/bin/chimera`, which is the path that the program calls Chimera Software.

All options are not forced to be, but it is highly recommended to use them, specially those related to the input/ouput files.

3.3 Logging

Activating the option for verbose (-v, --verbose), it prints to the standard error the log of the program. The point that are verbosed and the message if it is all right are the followings:

- Warning for files that were not analyzed because they were wrong.
- Print how many files are going to be analyzed.
- Prints if the input files are all homodimers, all heterodimers or a mix between homodimers and heterodimers and how many interactions there are.
- If there are no files because of any problem, it advise you to revise the pairwise interactions.
- Prints the percentage of identity of the best alignments and RMSD of the superimpositions.
- Prints where are the results stored, the output file.
- Tells you when the program finishes successfully.

3.4 Running the program

To run the program in the terminal:

```
$ python3 main.py -i [input files] -o [output file]
```

You can activate as many options as you would like:

```
$ python3 main.py -i [input files] -o [output file] -v
```

Chapter 4

Analysis of examples

4.1 Tested examples

We tested two examples. One which is an heterotrimer (heterotrimeric G protein) and one repeated dimer (the example provided by Python teacher).

4.1.1 Heterotrimer

The example used is an heterotrimeric G protein, whose PDB id is (3AH8). We split it into two subunits (chain A - chain B and chain B - chain G). These are the subunit for the heterotrimer:

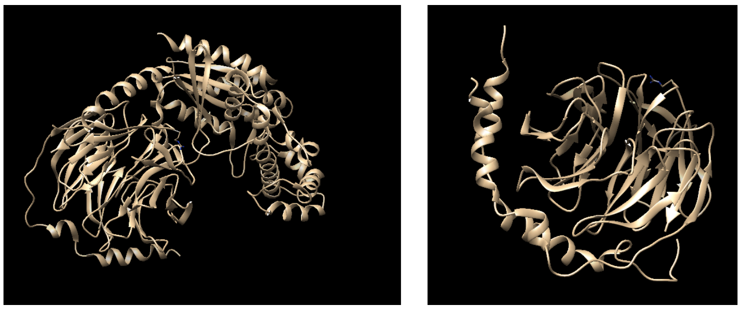

Figure 4.1: Subunits by pairs on a heterotrimeric G protein (3AH8). On the left, interaction of chains A-B. On the right, interaction of chains B-G.

After running the program, the heterotrimeric G protein looks like que original PDB file. RMSD is 0, because is a toy model and fits perfectly.

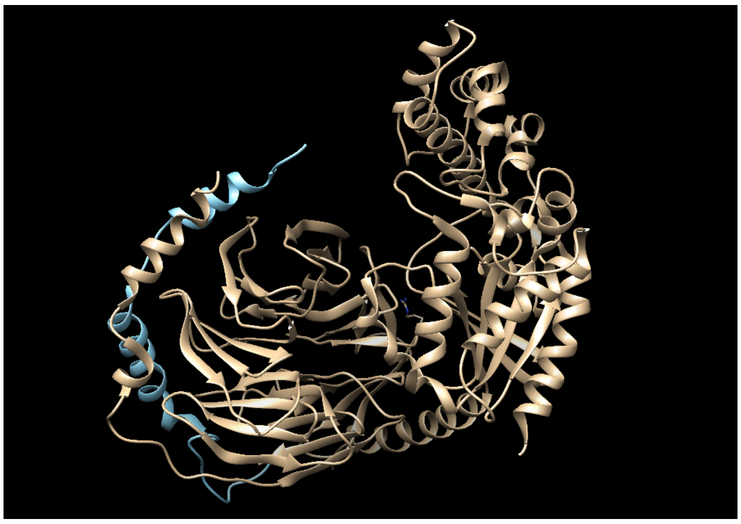

Figure 4.2: Complete structure of heterotrimeric G protein (3AH8), after running the program.

4.1.2 Homodimer

The example used for homodimers is the one provided by Professor Javier Garcia. This structure represents a nucleosome (and we were given the PDB files for interacting pairs of chains and those chains separatedly. We only use those interacting PDB files and NOT separated chains. There were 23 elements, homodimers and heterodimers repeated. Some of these interactions are represented in Figure 4.3.

Figure 4.3: Two examples of input files out of 23.

After running our program the final structure was the following (Figure 4.4).

His3 is a gene of a nucleosome. Figure 4.4 seems to be a complex nucleosome. At least, it has the proper shape. Therefore, although we do not know the real output, we can conclude that the program worked well.

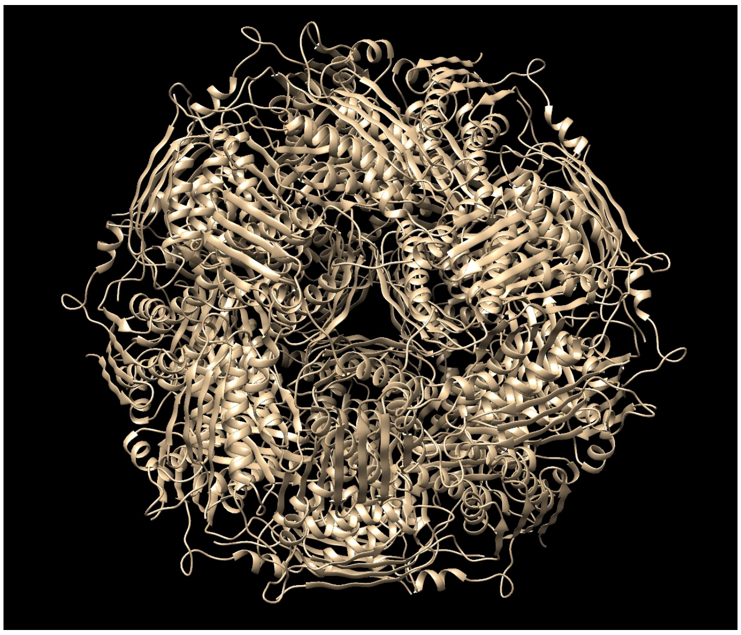
Figure 4.4: Final structure after running the program with the subunits given.

4.2 Generalisation of the program

Generally, the program will work perfectly for non-repeated/normal heterodimers and heteropolimers defined as it was done.

For homodimers or repeated heterodimers it is more difficult, because a structural superimposition will not rebuild the whole structure. Moreover, we cannot know the correct orientation nor the stoichiometry. For these reasons, the program may fail recontructing this kind of structures.

Chapter 5

Discussion of the project

In the project we had to understand and program usual algorithms in bioinformatics, such as sequence alignment and structural superimposition. It was a project that we were only told the title and we had to make up all the pipeline, the workflow, and the program. If we were in a lab, we had a real problem, with a real project and with real data (data for training and data for testing). In this project, we had not had data to train or test and for these reason we may think that our program works perfectly and actually it could not be like this. With our own examples it worked, the program finished and created well the output, but at this point there is no guarantee that the program will work for any protein, with any input files. Personally, we believe that our program does what we want, we tested it and it worked and the pipeline is coherent and theoretically it must work.

Chapter 6

Conclusions

This project have helped us to understand better concepts about structural superimposition, sequence alignment and python programming. In addition, we had to do the effort for create our own data and learn to use it. We did our best and we hope the program works.