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:

- All PDB file names which contains the interacting pair chains separated by commas.
- 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 pair interaction 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. If not output file is indicated, the program will asign different names depending on the inputs (homodimer, heterodimer or both), in the current directory or the one specify in the input (homodimer_complex.pdb, heterodimer_complex.pdb or homodimer_heterodimer_complex.pdb if mixed).

2.2 Modules and Packages

Biopython is the principal package used, as well as *sys*, *os*, *re*, *argparse*, *subprocess* and the self-created modules *common_functions*, *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 perform a global alignment of two sequences.

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 work with directories, for example, it can 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 files, searching the extension of 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 build 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*. 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*. Finally, if the pair of chains in the interacting files contains both homodimers and heterodimers interactions (A-A, A-B), then it is considered a mixed complex. For homodimers, repeated heterodimers and mixed complex, 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_homodimer_heterodimer, that is used when the input files are homodimers and heterodimers or repeated 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 with all 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 get_pdb_info

This function analyze all the PDB files from the input. It checks if all inputs are pdb pair interaction files. It extracts and compare the structures and their sequences in order to know if we have heterodimers o homodimers. It returns all pairwise interactions, the homodimer interactions, the heterodimer interactions and the wrong files that are not analized.

2.3.4 common_functions.get_sequence

This function deletes heteroatoms and returns the sequence of the proteins with more than 30 aminoacids (to avoid ligands). It uses CaPPBuilder package.

2.3.5 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 0.99 it returns True, else, it returns False. It uses Bio.pairwise2 package to make the alignments.

2.3.6 common_functions.chains_comparison

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

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 if a chain produces a clash in the new structure. 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 to the complex.

2.3.9 common_functions.save_complex

This function saves the new 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 in the structure, performs a pairwise alignment and stores the ids of those alignments with a percentage of identity (%id) greater than 99%, that are assumed to be the same chain 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, and the other as moving chain. Then the coordinates for the new interaction are updated and the chain used for superimposition is deleted in order not to have repeated superimposed chains. After the superimposition, it use *common_functions.clash_identifier* in order to add the new chain, and 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 the first one as reference and correcting the coordinates from the others, and afterwards check if there is any clash with *common_functions.clash_identifier* in order to add the new chain and not to have redundancies. It returns the built structure.

2.3.13 homodimers.get_structure_homodimer_heterodimer

This function is called when there both types of interactions, homodimers and heterodimers, in the input files. First, it superimposes homodimers interactions like function *homodimers.get_structure_homodimer* to build a dictionary with the superposed chains, and after that it adds heterodimers interactions within them, by aligned and superimposing them like heterodimers module. Finally, it check if there is any clash when adding a new chain with *common_functions.clash_identifier*. 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), distinct heterodimers (A-B, B-C...) or both types of interactions (A-A, A-B). To do that, we use *get_pdb_info* function, which implements the functions *get_sequence*, *seq_comparison* and *chains_comparison.*

If the interactions are homodimers 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 created in the modules using *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) If the option (-v, --verbose) is activated, it return the log of the program in the standard error.

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 the program only works with entered interactions. We do not consider the stoichiometry (how many repetitions can happen) and we assume that, the whole complex is composed by the input interaction. 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 only checks if the pdb file is a pair interaction, it does not check if they 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 30 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 models. This is done because Chimera has problems recognizing differet chains with the same name or ids different from A-Z.

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 Installing the program as a package

The program can be installed as a package. To do that, first unzip the project with the command

```
tar -xzfcv complex_reconstr.tar.gz
```

Once it is unzip, change the working directory to the folder and inicialize setup with the command

```
sudo python3 setup.py install
```

With this action, all modules and the main program will be installed as a package in Python Library.

3.3 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.

3.4 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, as well as they are not pdb files or pair interaction files.
- Print how many and which 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 wrong inputs, it breaks the program and 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.

3.5 Running the program on the terminal

To run the program in the terminal:

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

You can activate as many options as you would like:

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

If you want to run the program in the terminal, make sure that all modules are in the same directory than the *complex_reconstruction.py* script.

Chapter 4

Analysis of examples

4.1 Tested examples

We tested for examples. One which is an heterotrimer (heterotrimeric G protein), a big homodimer and two proteins with heterodimer and homodimer interactions

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:
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.

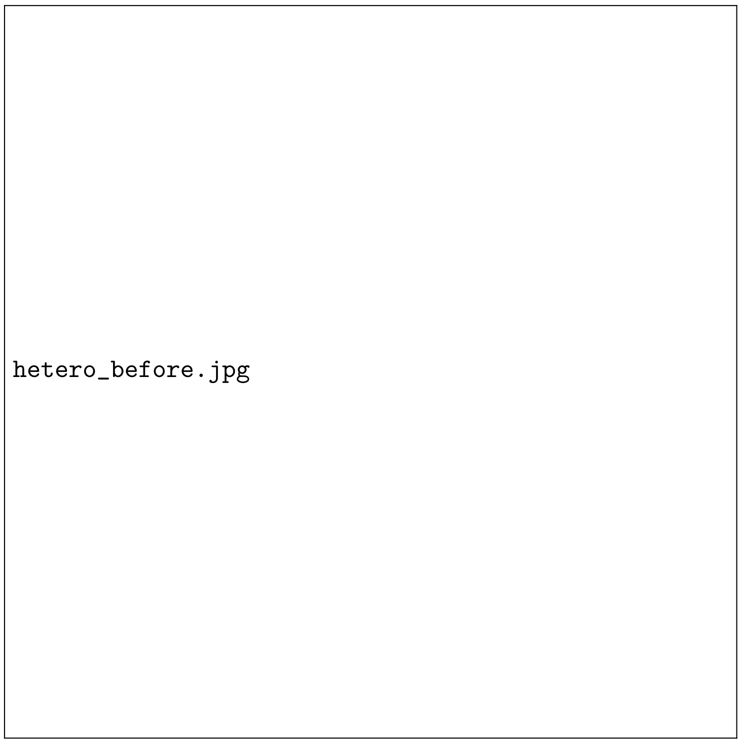

Figure 4.1: Subunits by pairs on a heterotrimeric G protein (3AH8). On the left, interaction of chains A-B. On the right, interaction of chains B-G.

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.

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

4.1.3 Homodimer and Heterodimer mixed

In this section we will show two examples of mixed homodimer and heterodimer interaction: one where the program worked, and one where the program did not work.

The first example used for this type of interactions is a hemoglobine (1GZX). From which we have 3 pair interaction files where two differnt chains are involve. These interactions are represented in Figure 4.5.

After running our program the final structure was the following (Figure 4.6).

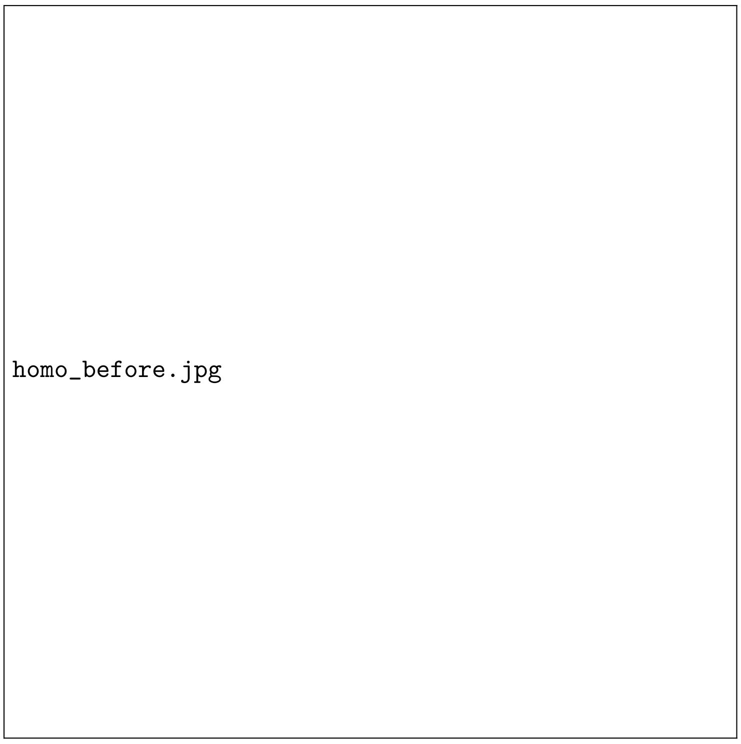

Figure 4.3: Two examples of input files out of 23.

As we can observe, the built complex obtained in Figure 4.6 seems to be a complex hemoglobin. At least, it has the proper shape. Therefore, we can conclude that the program worked well.

The next example used for this type of interactions is a viral capside formmed by 6 interaction files (3J7L). From which we have 4 homodimer interactions and 2 heterodimer interactions. These interactions are represented in Figure 4.7.

After running our program the final structure was the following (Figure 4.8).

As we can observe, the build complex obtained in Figure 4.8 seems not to be a viral comples. That is because, the program have only build the complex with the given chains, and to build a good one, we need all the repeated interactions. This a limitation of our program commented in section *Restrictions and Limitations of the program.*

4.2 Generalisation of the program

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

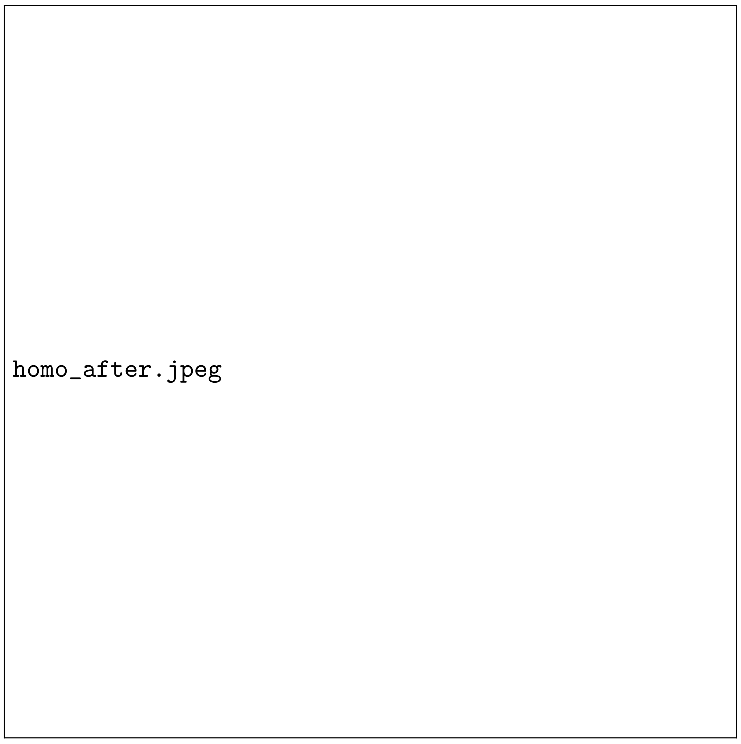

Figure 4.4: Final structure after running the program with the subunits given.

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

mix1_before.jpg

Figure 4.5: Three subunit pairs from hemoglobin. PDB id: 1GZX

mix1_after.jpg

Figure 4.6: Final structure after running the program with the subunits from 4.5

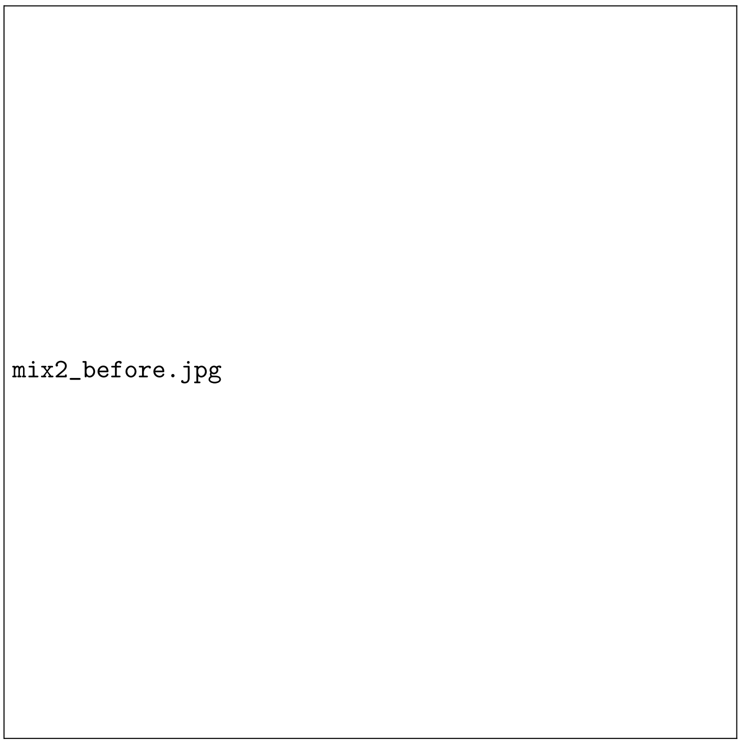

Figure 4.7: Subunits extracted from structure of a viral capside. PDB id: 3J7L

mix2_after.jpg

Figure 4.8: Final structure after running the program with the subunits from 4.7.

Chapter 5

Discussion of the project

In the project we had to understand and program usual algorithms in bioinformatics, such as sequence alignment, structural superimposition and protein-prootein. 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. We also had to decide what kind of interactions we can model or not. If we were in a laboratory, 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 real data to train or test and for these reason we may think that our program works correctly and actually it could not be like this. With our own examples it almost worked, the program finished and created well the output, but as we can see in the last example, 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.