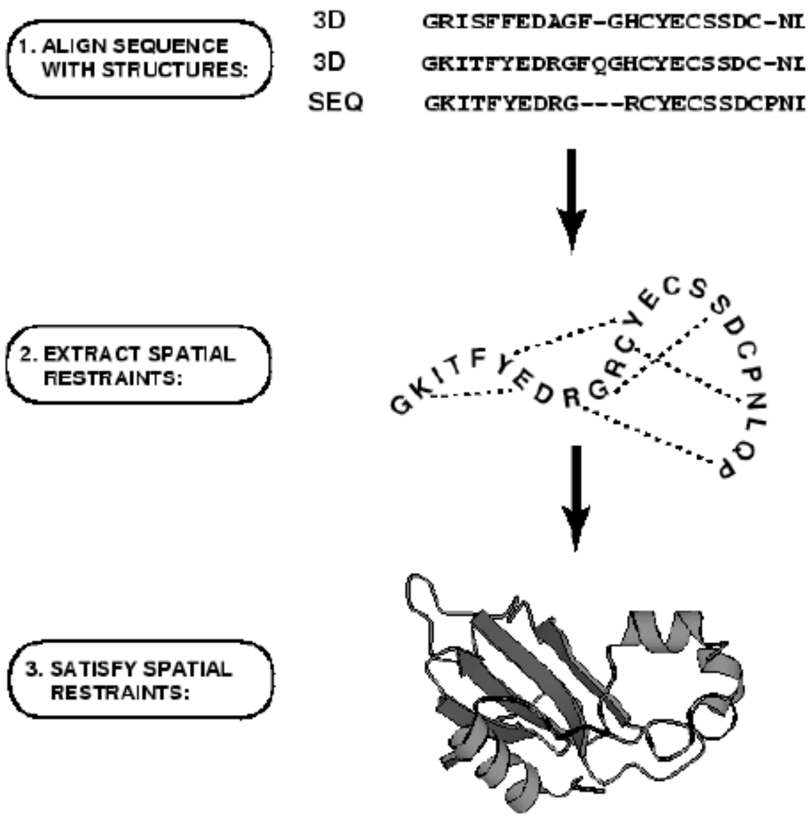
PRACTICE SBI 4P (SBI 7 + SBI 8 + SBI 9) COMPARATIVE HOMOLOGY MODELING

MODEL BUILDING

Nuria B. Centeno. Modified by B. Oliva
Model building by Modeller

Modeller is an implementation of an automated approach to comparative protein structure modeling by satisfaction of spatial restraints.
First, the known template 3D structures are aligned to the target sequence to be modeled. Second, spatial features, such as Ca-Ca distances, hydrogen bonds, and mainchain and sidechain angles, are transferred from the templates to the target. Thus, a number of spatial restraints on its structure are obtained. Third, the 3D model I obtained by satisfying all the restraints as well as possible:


1. ALIGN SEQUENCE WITH STRUCTURES:
3D GRISFFEDAGF-GHCYECSSDC-NI
3D GKITFYEDRGFQGHCYECSSDC-NI
SEQ GKITFYEDRG---RCYECSSDCPNI
2. EXTRACT SPATIAL RESTRAINTS:
GKITFYEDRGRCYECSSDCPNLQP
3. SATISFY SPATIAL RESTRAINTS:

Getting started

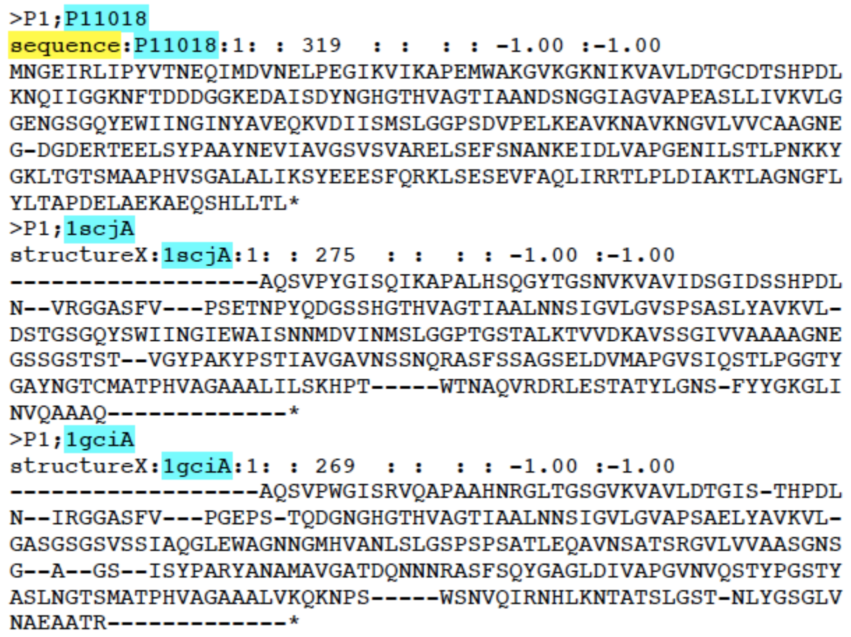
Download the target and example of a modeller input file from Aula Global

It is advisable to create two different directories: one for the models we will obtain from our sequence alignment and the second one for the models we will obtain from our alignment based on the structural alignment of our templates. In each directory we will need a copy of the corresponding alignment, the pdb files for the templates and the modeller input file.

Using MODELLER
Before running the program, we will need to do the following steps:

1. Convert the alignment into pir format
The alignment used by modeller must be in pir format. We can use the program **aconvert** to do this format change.
The input format depens on how we have obtained the alignment: "c" if it comes from clustalw and "h" if it comes from hmmer.
The output format is pir ("p"). The resulting alignment will have as many blocks as sequences we have in our alignment. Each block has a header of two lines with some labels, which must be the same as in the input file (see below).

The file will look like:



```
>P1;P11018
sequence:P11018:1: : 319   : :   : : -1.00 :-1.00
MNGEIRLIPYVTNEQIMDVNELPEGIKVIKAPEMWAKGVKGKNIKVAVLDTGCDTSHPDL
KNQIIGGKNFTDDDGGKEDAISDYNGHGTHVAGTIAANDSNGGIAGVAPEASLLIVKVLG
GENGSGQYEWIINGINYAVEQKVDIISMSLGGPSDVPELKEAVKNAVKNGVLVVCAAGNE
G-DGDERTEELSYPAAYNEVIAVGSVSVARELSEFSNANKEIDLVAPGENILSTLPNKKY
GKLTGTSMAAPHVSGALALIKSYEEESFQRKLSESEVFAQLIRRTLPLDIAKTLAGNGFL
YLTAPDELAEKAEQSHLLTL*
>P1;1scjA
structureX:1scjA:1: : 275   : :   : : -1.00 :-1.00
------------------AQSVPYGISQIKAPALHSQGYTGSNVKVAVIDSGIDSSHPDL
N--VRGGASFV---PSETNPYQDGSSHGTHVAGTIAALNNSIGVLGVSPSASLYAVKVL-
DSTGSGQYSWIINGIEWAISNNMDVINMSLGGPTGSTALKTVVDKAVSSGIVVAAAAGNE
GSSGSTST--VGYPAKYPSTIAVGAVNSSNQRASFSSAGSELDVMAPGVSIQSTLPGGTY
GAYNGTCMATPHVAGAAALILSKHPT-----WTNAQVRDRLESTATYLGNS-FYYGKGLI
NVQAAAQ-------------*
>P1;1gciA
structureX:1gciA:1: : 269   : :   : : -1.00 :-1.00
------------------AQSVPWGISRVQAPAAHNRGLTGSGVKVAVLDTGIS-THPDL
N--IRGGASFV---PGEPS-TQDGNGHGTHVAGTIAALNNSIGVLGVAPSAELYAVKVL-
GASGSGSVSSIAQGLEWAGNNGMHVANLSLGSPSPSATLEQAVNSATSRGVLVVAASGNS
G--A--GS--ISYPARYANAMAVGATDQNNNRASFSQYGAGLDIVAPGVNVQSTYPGSTY
ASLNGTSMATPHVAGAAALVKQKNPS-----WSNVQIRNHLKNTATSLGST-NLYGSGLV
NAEAATR-------------*
```

2. Modeller input file (extension .py)
The input file contains all the parameters needed by MODELLER and the options we want to use for building up the 3D models of our target. It is advisable that this file has the extension .py

It will look like:

```
# Homology modeling with multiple templates from modeller import *
# Load standard Modeller classes from modeller.automodel import *
# Load the automodel class log.verbose()
# request verbose output

env = environ() # create a new MODELLER environment to build this model in

# directories for input atom files

env.io.atom_files_directory = ['.', '../atom_files']
a = automodel(env,
alnfile = 'P11018_1scjA_1gciA.pir', # alignment filename
knowns = ('1scj', '1gciA'), # codes of the templates
sequence = 'P11018') # code of the target
a.starting_model= 1 # index of the first model
a.ending_model = 2 # index of the last model

# (determines how many models to calculate)
a.make() # do the actual homology modeling
```

Therefore, we need to modify the input file, which is only an example, to incorporate our data:

a. In **alnfile** we will set the name of our alignment file (in pir format) from which we will build up the models. The name of the file is written between single quotation marks.

b. In **knowns** we will set the label for each template. These labels must be consistent (i.e the same) with those appearing in the alignment file (first and second line of each template block) and with the prefix of its pdb file. Template codes are written between single quotation marks,and they are spaced by commas and a blank space.

c. In **sequence** we will set our target label. This label must be consistent with the one appearing in the alignment file (first and second line of the target block). It is written between single quotation marks.

d. By default pdb files must be in our working directory. If not, we need to state its location at **env.io.atom_files_directory**.

e. We can build up as many models as we want, since there is more than one solution satisfying the spatial restrains. In our case, we will only build up two models for each alignment. To do so, we will set 2 in

a.ending_model. The instruction **a.make()** is for doing the homology modeling in its most simple formulation.

f. Modeller can work with several chains. However, the simplest form is to use a single chain for templates. In order to get the structures of single chains we run the script PDBtoSplitChain.pl

```
PDBtoSplitChain.pl -i <PDB-file> -o <root-name>
```

Where the output will be as many files as chains in the PDB file with the PDB and FASTA files of the chains (i.e. for a PDB with chains A and B, using a root name "root" we get rootA.pdb, rootB.pdb, rootA.fa and rootB.fa files)

The command for running MODELLER is:

```
$mod9.13 file
```

where file is the file with extension py that we have just modified.

However, in the next steps we plan to include python commands of MODELLER. This implies the use of specific libraries that we may wish to apply. Consequently, we have a second option to run MODELLER, indicating what python we plan to use

```
$modpy.sh  python file
```

Loop modeling, refinement and assessment

The input file uses a standard procedure (it's actually a python class) "automodel" for modeling. However, we can improve the modeling of loops by using the class "loopmodel" .

a. Change automodel by loop model in the input file

    a=loopmodel(

b. Include the assessment by adding a new parameter within the parenthesis

    …, assess_methods=assess.DOPE

c. Sort the output according to DOPE energy. After the execution of modeling (i.e. a.make()), you have to add the following lines:

```
ok_mdl=filter(lambda x:x[‵failure’] is None,a.outputs) #remove fail
key=’DOPE score’
ok_mdl.sort(lambda a,b: cmp(a[key],b[key]) #sort OK models
#list the ranking
for m in ok_models:
    print(“Model: %s  (DOPE score %.3f)”%(m[‵name’],m[key]))
```

d. Refinement. This can be done at two levels: 1) refining the whole model; and 2) refining only the loops.

   a. To refine the whole model add a line after the definition of the number of ending model
      ```
      a.md_level=refine.fast
      ```

   b. To refine the loop add three more lines
      ```
      a.loop.starting_model=1  #First loop model as *.BL0001
      a.loop.ending_model=4   #Last loop model as *.BL0004
      a.loop.md_level=refine.fast
      ```

      However, if we wish to evaluate also the assessment of the loops, we add in the execution, instead of assess_methods, loop_assess_methods. Then, we report in the ok_mdl dictionary the selection of acceptable a.loop.outputs, instead of a.outputs.

e. In case we wish to have only a selected set of loops to modify, then we need to use restrictions. Therefore we have to define our class, this being a child of loopmodel. For example, if we wish to select residues 19-28 and 45-50 from the model, we define MyLoop class as:

   ```
   Class MyLoop(loopmodel):
       def select_loop_atoms(self):
               return selection(self.residue_range(’19:’,’28:’),
                                   self.residue_range(’45:’,’50:’))
   ```

   Then we run MyLoop class instead of loopmodel class.

Macro-complex modeling and restrictions

a. Modeller also handles the modeling of a complex with more than one chain. The class is the same (automodel), but the alignment needs

to handle with the different behavior.
The alignment places a slash "/" to show the end of a chain. Then it gives the residue position at the start and end of the template(s) (i.e. 1:A and 74:B). If more than one chain is to be built, see the following example:

```
C; example for building multi-chain protein models

>P1;2abx
structureX:2abx:   1 :A:74 :B:bungarotoxin:bungarus multicinctus:2.5:-1.00
IVCHTTATIPSSAVTCPPGENLCYRKMWCDAFCSSRGKVVELGCAATCPSKKPYEEVTCCSTDKCNHPPKRQPG/
IVCHTTATIPSSAVTCPPGENLCYRKMWCDAFCSSRGKVVELGCAATCPSKKPYEEVTCCSTDKCNHPPKRQPG*

>P1;1hc9
sequence:1hc9:   1 :A:148:B:undefined:undefined:-1.00:-1.00
IVCHTTATSPISAVTCPPGENLCYRKMWCDVFCSSRGKVVELGCAATCPSKKPYEEVTCCSTDKCNPHPKQRPG/
IVCHTTATSPISAVTCPPGENLCYRKMWCDAFCSSRGKVVELGCAATCPSKKPYEEVTCCSTDKCNPHPKQRPG*
```

b. However, in many occasions there is symmetry between both chains (in case of an homodimer). Constraints are applied by defining a new class inheriting the automodel

```
class MyModel(automodel):
    def special_restraints(self, aln):
        # Constrain the A and B chains to be identical
        # (but only restrain
        # the C-alpha atoms, to reduce the number of interatomic
        # distances that need to be calculated):
        s1 = selection(self.chains['A']).only_atom_types('CA')
        s2 = selection(self.chains['B']).only_atom_types('CA')
        self.restraints.symmetry.append(symmetry(s1, s2, 1.0))
```

c. For a more general use of restraints, we can add as many new restraints as we wish, forcing specific distances between CA atoms (or any specific selected atom) of residues. Restraints are stored in the set "restraints" within automodel. They can be specific of secondary structures (alpha or beta) or any specific distance forced by the user. See the following example:

Create the class

```
class MyModel(automodel):
    def special_restraints(self, aln):

#     Define the sets

        rsr = self.restraints
        at = self.atoms
```

Use a file with restraints if available

```
#       Add some restraints from a file:
#       rsr.append(file='my_rsrs1.rsr')
```

Define secondary structure restraints

```
#       Residues 20 through 30 should be an alpha helix:
        rsr.add(secondary_structure.alpha(self.residue_range('20:', '30:')))


#       Two beta-strands:
        rsr.add(secondary_structure.strand(self.residue_range('1:', '6:')))
        rsr.add(secondary_structure.strand(self.residue_range('9:', '14:')))

#       An anti-parallel sheet composed of the two strands:
#        Requires the location of the starting Hbond and the total of residue-pairs
#        that will form the ladder witha - sign
        rsr.add(secondary_structure.sheet(at['N:1'], at['O:14'],
                                          sheet_h_bonds=-5))
#       Use the following instead for a *parallel* sheet:
#        Requires the location of the starting Hbond and the total of residue-pairs
#        that will form the ladder witha + sign
#       rsr.add(secondary_structure.sheet(at['N:1'], at['O:9'],
#                                         sheet_h_bonds=5))
```

Use your own selected restraints

```
#       Restrain the specified CA-CA distance to 10 angstroms (st. dev.=0.1)
#       Use a harmonic potential and X-Y distance group.
        rsr.add(forms.gaussian(group=physical.xy_distance,
                               feature=features.distance(at['CA:35'],
                                                         at['CA:40']),
                               mean=10.0, stdev=0.1))
```

Possible error sources when executing modeller

The most frequent error sources while executing MODELLER are three:

1. File names and their location
2. Inconsistencies between labels stated in the alignment and input files.
3. Inconsistencies between sequences in the alignment and in the template files. This may occur in different situations:

    a. if the sequence used in the alignment was obtained from Swissprot rather than from the PDB. Usually the sequence at the pdb is not the complete sequence.
    b. if in the protein structure there is one main chain segment with more than one position (occupacy < 1). In that case, the program **PDBtoSplitChain** interprets that we have different aminoacids instead of one aminoacid with different positions,

making a mistake while extracting the sequence. In this case, in the log file will appear an error like this:

```
read_te_291E> Sequence difference between alignment and pdb :
x (mismatch at alignment position 258)
Alignment TTTKLLGGDSFYYGKGLINVQAAAQ
PDB       TTTKLGDSFYYGKGLINVQAAAQ
Match     *****        *       *
```

Once we have identified the flexible zones, we need to manually modify the FastA file with the correct sequence (eliminating duplicities) and obtain a new alignment. However, in this practice, due to time reasons, we can just eliminate this kind of templates.

Once we have successfully executed the program, we will obtain a series of output files. All of them will have as prefix the label we set in the **sequence** option. The important files for us are those containing the three-dimensional coordinates of the generated models in pdb format, which will have as extension **.B9999000N.pdb**, where N is the number of the different models we have generated (1 and 2 in our case).

It is really important to change the name of these files (sequence.B9999000N.pdb) to simpliest and more meaningful names. Keep in mind that if we execute again MODELLER in the same directory, these files will be overwritten.

After executing MODELLER with the two alignments we have, we will have four models: two coming from the sequence-based alignment and two more coming from the alignment based on the templates structural alignment.

At this point, it is worth to visualize the obtained models and even superimpose their structures to realize similarities and differences between them. It is also worth comparing our models with the templates from which we have obtained them.

In some cases, this kind of analysis may prompt us to refine the used alignments. Two examples would be:

1. We observe a loop located in such a way that it breaks a secondary structural element. This is an indication that the alignment has opened a gap without taking into account the structural information of the templates. This is more likely to happen in a sequence-based

alignment.

2. We observe that in the N-term and/or C-term of the model there are disordered segments. This is because there are gaps at the beginning and/or the end of the alignment. In this situation, it is advisable to build up a model without these segments. On the other hand, the presence of loops with different conformation from the one observed in the templates, are indicative of gap regions in our alignment. These zones, in which we have less information to build up the model, deserve a close attention in our analysis.