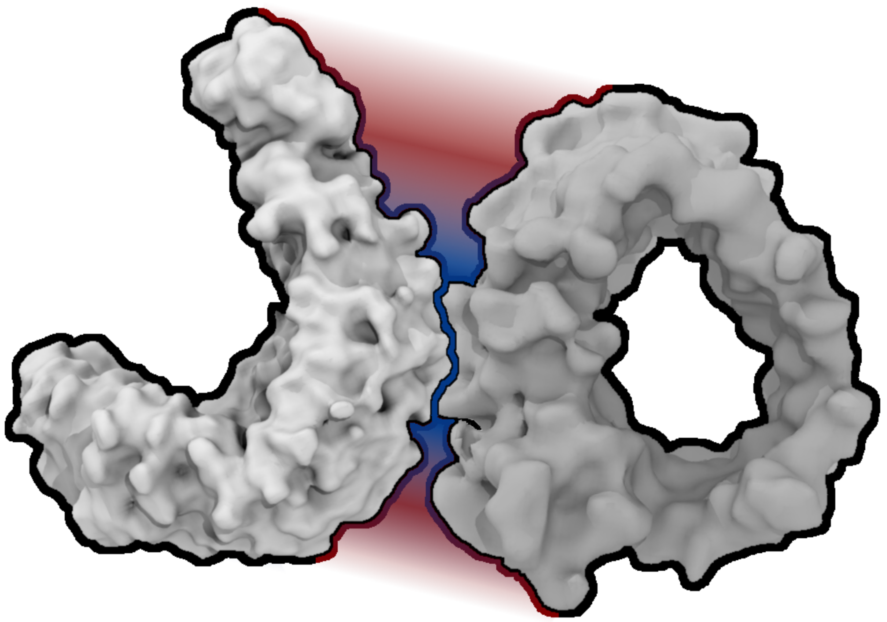
# JabberDock 1.0 Tutorial

A brief tutorial on using the JabberDock protein-protein docking software

Lucas S.P. Rudden & Matteo T. Degiacomi

Durham University, September 2019



# Contents

## 1 Introduction

This tutorial is designed to get you up to speed with using JabberDock and will put particular emphasis on easing those with limited computational biophysics experience. Please do not be put off if you are from a structural biology background and have not tried command-line software before. Equally, if you are familiar with the ideas and concepts here, free to skip parts that aren't necessary. If you require more specific information on what you can achieve, or what the JabberDock Python module provides, please refer the manual (in the main folder).

We will be using the bound version of a protein used in our JabberDock unbound benchmark [1], the ribonuclease inhibitor - ribonuclease A dimer (PDB: 1DFJ). The two structures provided to you have been cleaned up and should work without any additional effort (see 1.2). You may follow this tutorial with any pair of proteins you like, however, there are a few issues you might encounter if you blindly go into a docking exercise. Some of the most common issues and ways to fix them will be discussed in this tutorial.

We recommend you run the docking part of this on a supercomputer. Figure 1 shows how the mass of a complex scales with the time taken to perform a job, so while it is possible to run on a desktop depending on what hardware you have, it might take a while. We are currently working on reducing hardware requirements in future iterations of JabberDock.

JabberDock should be run on a terminal in a Unix-based operating system. There is no current Windows equivalent. In principle, you can import the JabberDock Python module and create your own Windows equivalent commands, or just a Python script to run, if you wish to use JabberDock anyway. The software you need before using JabberDock are discussed in the manual and the README file.

A full JabberDock run is divided into four steps:

- Generating a short molecular dynamics (MD) run on both the receptor and ligand
- Creating the Spatial and Temporal Influence Density (STID) maps to represent the proteins in the docking environment
- Performing the docking using the STID maps of both binding partners
- Ranking the resulting complexes

Each of these steps features specific input parameter and output files. The command `jabberdock.py` executes the four steps automatically by using default settings (see Section 2).

The advanced user can customize the behaviour of JabberDock by executing each of these steps independently via dedicated commands, namely `gmx_run.py`, `build_map.py`, `dock.py` and `rank.py`. More information on each command and the appropriate flags can be found by typing the command on the console followed by `-h`. A step-by-step tutorial on the usage of these scripts is presented in Section 3.

### 1.1 Hardware

Before we begin discussing how to use JabberDock, we will briefly outline the key pieces of computing hardware JabberDock requires (in layman terms).

- CPU (Central Processing Unit). This performs the basic tasks that run your computer and is responsible for the general running of programs. JabberDock runs most of its tasks in parallel, i.e. dividing calculations between multiple CPUs. Put simply, the more CPUs you have, the better. For instance, if each CPU analyses one docking position, then having two working at the same time will mean you can approximately half the time taken to perform your calculations. This is where a supercomputer can come in handy, as they have many CPUs that you can split your jobs across. Figure 1 shows how the number of CPUs and time taken to perform the calculations scale with mass of a protein complex scales 1.

[1] Rudden L.P., Degiacomi M.T., Protein docking using a single representation for protein surface, electrostatics and local dynamics, JCTC, 2019

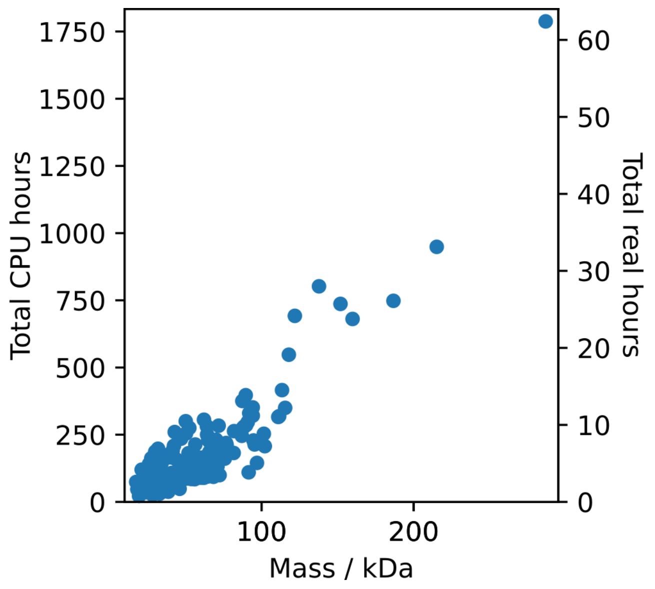


**Figure 1:** *Relationship between the mass of the total protein complex simulated with JabberDock versus the time taken to run the entire POW optimisation search. The runs were made across 28 CPUs, the architecture is discussed in the Computational Load section in the supplementary details of JabberDock article. The left y-axis displays the total time to compute per CPU, whereas the right y-axis indicates the real-time it took across the 28 CPUs.*

- GPU (Graphics Processing Unit). This renders everything, from your desktop to YouTube videos. In other words, it's what you "see" on your screen. They can be used to perform very simple arithmetic, and when used right can both speed up and cheapen the cost of a calculation. Most motherboards in a desktop PC come with an onboard GPU chip in case you don't have a dedicated one. You will need a dedicated GPU if you want to use it properly, so please check what hardware you have before using one (you will just get an error if you try otherwise).
- RAM (Random-Access Memory). This temporarily remembers information needed for tasks, such as arrays of numbers, very quickly (a hard drive is used for long term memory storage, and are much slower but larger). Whenever you're evaluating a docked pose, RAM is required to hold information about the protein's structure and position, such as its coordinates. In other words, the more CPUs you're asking for, the more memory you will require, so there is a trade-off to be aware of. We provide specific information regarding the architecture of the machines we used in the supplementary details of our article.

## 1.2 Selecting and preparing the docking candidates

What do we mean by "cleaning the docking candidates"? Well, when you download a structure from any PDB database, you are downloading the atomic structure in the state it was crystallised, assuming it's been sourced from x-ray crystallography (currently most commonly the case). These often contain solvents used to aid the crystallisation process and non-standard amino acids. Or, perhaps you're interested in how a monomer unit from a homodimer docks into some other protein, but you don't need to simulate the entire homodimer?

While there is some evidence that simulating the protein in its oligmeric form before cleaving out the monomer unit is beneficial, by cleaning the structure, we essentially mean removing the parts we don't need or want - i.e. the solvent molecules, the other monomer unit of the homodimer, etc. What we want is just the 20 standard amino acids assembled into the protein candidate you're interested in. You can remove the unwanted solvent by simply deleting them from the text file, and the desired monomer unit can be selected using standard atomselect commands in VMD. For example, suppose you have determined that the chain B within a trimer is your docking candidate. Open the molecule up with VMD and select the molecule by clicking on its name underneath "molecule". Then right-click on it and click "save coordinates". A "Save Trajectory" window should pop up. In the "Selected atoms:" window, type: "chain B" and click save. This copies just the structure you need as a new file.

You will find out whether there is an issue with a pdb when you attempt to run GROMACS. In this tutorial we provide you with two pre-prepared structures to work with.

## 2 Running JabberDock automatically

There is a shortcut command you can use to skip all of the individual steps, `jabberdock.py`. This command will run all calculations using the default settings. To run this command, all we need to type is:

```
jabberdock.py -ir 1dfj_0.pdb -il 1dfj_1.pdb -ff amber03 -np 4 -gpu 0000 -ntomp 4 -
ff_dat /home/username/biobox/classes/amber03.dat
```

The `-ir` flag defines the receptor pdb file, in this case, `1dfj_0.pdb`, one of our test structures. `-il`, in turn, specifies our ligand, `1dfj_1.pdb`. Try to always use the smaller structure as the ligand — this helps speed up the docking phase. `-ff` refers to our desired forcefield, here it's `amber03`. A longer discussion on what a forcefield is and what we're referring to more specifically can be found in Section 3.1, `amber03.ff` is present in the topology file where GROMACS is installed.

`-np` details the number of processes you want to run in parallel, both for the GROMACS run and the docking, `-gpu` specifies the GPU id to map the GROMACS calculations to, and `-ntomp` is the number of threads you want to run for the GROMACS MD. More information on all of these is provided in Section 3.1. The `-ff_dat` flag refers to the data file that contains our charge data used for the GROMACS run. More information on this, as well as how you can create your data files outside the `amber03` one provided, is discussed in Section 3.2.

Running this executable will generate all of the GROMACS intermediate steps and density maps necessary for docking. It will also produce a folder called models, in which you will find all the docked predicted structures. Specifically, you're interested in the `initial_1dfj_0_sim_map.pdb` file coupled with any of the `model_x.pdb` files. `ranked_scores.dat` contains our ranked results, so the top model is our top-scoring result, the second number in the row is the model number (so the `x` in `model_x.pdb`). We discuss in greater depth the files generated in each of the steps listed below and provide some assistance with the Linux side of preparing your docking routine.

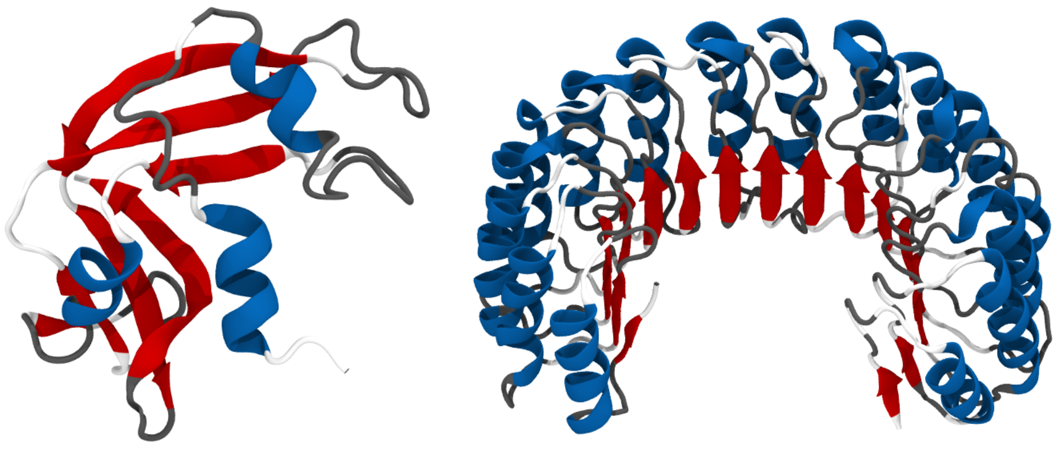


**Figure 2:** *Our two starting pdb structures, 1dfj_0.pdb on the left and 1dfj_1.pdb on the right.*

# 3 The JabberDock commands - fine tuning your docking

## 3.1 Sampling protein local dynamics

In this section, we will take our two proteins (shown in Figure 2), and perform an MD simulation on them using the GROMACS MD engine. MD is a widely used method to sample the conformational space of molecules.

### 3.1.1 What is an MD simulation?

Here, we will only briefly discuss it in case you have never encountered it before. In short, we begin by taking a molecular structure (typically a PDB atomic model), and assign what we call a forcefield to the structure. This means that each atom is given associated properties, such as an electric charge and van der Waals radius, as well as connectivity with its neighbours (depending on its covalent bond network). This information determines how every atom will interact with its surroundings. Interactions are classified as non-bonded (electrostatic and van der Waals) and bonded (typically bond, angle and dihedral), and their respective strength is determined by parameters specifically tuned to ensure a simulation reproduces experimental observables.

The sum of all interactions on an atom will impart a force it. Using Newton's equations of motion we can then calculate how an atom will move in time, i.e. based on its mass, simulation time step, and resultant force acting on it. This website provides an excellent course on what MD is, and how powerful a tool it can be to use in your research: erastova.xyz

GROMACS does all this for you after you have provided your structure and forcefield. In principle, depending on what type of simulation you are running, you'll need to supply temperature, pressure and a few other conditions, but since the conditions in the body are fairly fixed, we assume for the most part you are running within an environment of 310.15 K and 101 kPa. You can change these settings when running JabberDock, please refer to the manual for more details.

Assuming you have successfully installed all the software you need for JabberDock (listed in the manual), we begin with our simulations. A typical MD run requires three separate phases: 1) Energy minimisation, 2) Equilibration and 3) Production. The minimisation relaxes the structure such that the atoms are not unphysically close, the equilibration performs a short MD on our molecule under the desired conditions, such that it (should) achieve thermodynamic equilibration. Finally, production is where we run the MD and record the trajectory for our density maps.

We highly recommend at least going through the first GROMACS tutorial before continuing at this point, as it will not only check whether you have it correctly installed, but it will also give you an idea as to what's going on behind the scenes.

### 3.1.2 Running MD simulations for JabberDock

Let's begin by creating a new folder somewhere and running everything from there. Open up a new terminal and move to wherever you want to run from typing:

```
cd folder_name/
```

Then, let's create a new folder:

```
mkdir jd_tutorial
```

Then move into the new `jd_tutorial` folder. Now, let's copy in our pdb files into `jd_tutorial` so that if something goes wrong, we still have the original data. This depends on where you have installed JabberDock, but let's say you installed it in your `home/username` folder (the folder open when you open a new terminal).

```
cp /home/username/JabberDock/tutorial/*pdb .
```

The `*` wildcard indicates that any file ending with pdb in the tutorial folder will be copied. Alternatively, you could specify `1fdj_0.pdb` and `1dfj_1.pdb` independently, but the wildcard method is quicker. Just be careful when using it, as it will pick up anything that matches that pdb description. The `.` means we're copying those two files into the current folder we're in, the `jd_tutorial` folder specifically — if you managed to successfully moved into it, otherwise, you'll be copying it elsewhere.

If `gmx_run.py` is in your `PATH` (i.e. your computer can see the `/home/username/JabberDock/auto_scripts` folder wherever you are), you can run the command easily from the command line. If not, you have two options here. You can either call `gmx_run.py` using the path to the directory, or you can add `auto_scripts` to your `PATH`. we recommend the latter, as it means less work. You might have to google this depending on what type of Unix shell you're using (e.g. csh, bash etc.). On bash, you could do the following. First move into your home directory, you can change directly in there, or you can just type:

```
cd
```

which should take you there automatically. Then, we want to open up our `.bashrc` file, essentially a list of commands to run every time you open up a new terminal. You can use any editing software, such as vim, for this bit. To keep things simple, we will use gedit:

```
gedit .bashrc
```

At the bottom of the file, add the following line:

```
export PATH=$PATH:/home/username/JabberDock/auto_scripts;$Path
```

The `/home/username` part is just where you have installed JabberDock. If you are not sure, move into the JabberDock folder and use:

```
pwd
```

to find out where the folder is located. Finally, either close and reopen your terminal, or type:

```
source .bashrc
```

Now you should be able to use the JabberDock commands such as `gmx_run.py` or `dock.py` anywhere on your computer. If you do not want to do this, or you cannot get it to work, you will have to include where the scripts are located in every command, such as:

```
/home/username/JabberDock/auto_scripts/gmx_run.py
```

instead of just `gmx_run.py`. Move back into the folder you put those pdb files. We can test if our JabberDock commands are working correctly by typing:

```
gmx_run.py -h
```

or the equivalent. The `-h` flag means help. If everything is running smoothly, you should see a summary of what the command is and the possible flags you can attach. The manual expands on this information, but this is a good, quick way of making sure you're using the right inputs. Let's begin running our MD:

```
gmx_run.py -i 1dfj_0.pdb -ff amber03 -np 4 -ntomp 4 -gpu 0000
```

So, what's going on here? The `-i` flag is indicating our input structure, in other words, we want to run an MD cycle on the `1dfj_0.pdb` structure in the current folder we're sat in. The `-ff` refers to the forcefield we want to use. Once you've installed GROMACS, have a look in GROMACS' topology folder. On my computer, this is located in `/usr/local/gromacs/share/gromacs/top` (the GROMACS folder might be elsewhere, but the top should always be in the same relative place). Inside here you should see several forcefields listed, such as `amber03.ff`, `charmm27.ff` and `gromos43a1.ff`. This `-ff amber03` specifies the `amber03.ff` forcefield, note that we omitted the `.ff`. This also corresponds to the `amber03.dat` file in the `biobox/classes` folder, which we will need when we create the docking maps.

Next, we have the `-np` flag. This concerns the number of processors we want to use. This is where checking what hardware you have installed is important. The `-ntomp` provides the number of threads, or tasks, to perform at the same time (in a similar vein to the parallel processing we talked about before). Finally, the `-gpu` flag indicates the GPU we want to perform our tasks on. If you don't have a dedicated GPU, this will fail (so just omit it). The zeros refer to the first GPU you have installed and will be the same one rendering your desktop (if this GPU is installed in your desktop PC and you're not running this remotely). If you are lucky enough to have more than one installed, we would recommend using a different one by calling `-gpu 1111` and so on. You can usually check what GPUs are free using:

```
nvidia-smi
```

We reference the GPU four times (4 zeros) to correspond to each of the 4 processors we're using. More information on these last three commands is available on the GROMACS website. Remember that to run across multiple cores, you will need a specific version of GROMACS installed (GROMACS mpi), again, please refer to the website for more details on this. If you don't want to run across multiple processors, just specify `-np 1`.

Let this run to completion, it should do all three of the steps we mentioned above automatically. When it is finished, take a look at `1dfj_0_sim.pdb` using VMD. Does it look reasonable? Now we have our simulation for the receptor, we can do the same for our ligand:

```
gmx_run.py -i 1dfj_1.pdb -ff amber03 -np 4 -ntomp 4 -gpu 0000
```

### 3.1.3 Common issues

Most of the problems you might encounter in this phase will be GROMACS related, so if you cannot find your answer here, then please have a look online. That being said, we will briefly discuss some of the most common issues you might face (you should not see any of these with the test models).

- "atom X is missing in residue YYY in the pdb file" — This means that (at least) one of your amino acids is missing an expected atom that normally makes up the structure, usually because they're not resolved. You should be able to use software like modeller or chimera to fix this. We just need all atomtypes corresponding to the correct residue in the rtp file to be present.
- "Residue XXX not found in residue topology database" — This probably means you've got some non-standard amino acids in your structure or you've got some solvent molecules still present in the structure that are not parameterised in the forcefield. In the case of non-standard amino acids, you'll either need a different forcefield that has parameters for them or find a different structure/switch the residues. If you have solvent molecules (e.g. $SO_4$), then deleting these at the end of the file is usually enough to fix the issue.

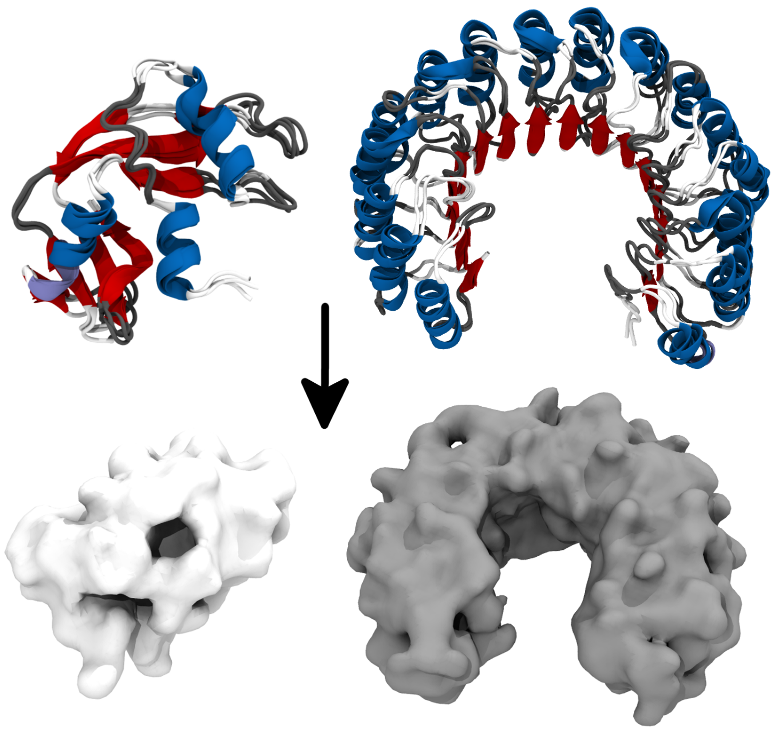


**Figure 3:** *Converting our MD simulations into the STID maps we need for docking.*

## 3.2 Generating the STID maps

Now we have our two simulations it is time we converted that into our Spatial and Temporal Influence Density (STID) maps, shown in Figure 3. We can do this with the following command:

```
build_map.py -i 1dfj_0_sim.pdb -ff_dat /home/username/biobox/classes/amber03.dat
```

Again, the `-i` flag refers to the desired input, in this case, it's our output from the first `gmx_run.py` simulation. `-ff_dat` provides the forcefield data file that contains the charge information we need to generate our maps. The `amber03.dat` file is shipped with the biobox version shipped with JabberDock, found in `/biobox/classes`. As part of the installation in the manual, we recommended moving biobox into your home folder (or wherever JabberDock is located), but we are still specifying the exact location of the `.dat` file just to be safe.

You will need a corresponding data file to whatever forcefield you used to ensure that the STID map is built from the correct charges. It is possible to produce your own using the `create_pqr_dat.py` function in `JabberDock.methods.data`. Let's say you used `amber99.ff` to run your simulation. If you open up a python terminal, you can run the following code (or similar) to generate your own pqr files (i.e. files containing information of charge and radius of every atom):

```
import JabberDock as jd

jd.data.create_pqr_dat(in_file =

                "/usr/local/gromacs/share/gromacs/top/amber99.ff/aminoacids.rtp",

                out_file = "amber99.dat")
```

This will put the new `.dat` file in your current working folder.

When running `build_map.py`, you might notice a warning about the residue HIS if your structure contains any histidines. This is because GROMACS keeps the label HIS when running its simulations, but the .dat files refer to the residues by their different protonation states: HID, HIE and HIP. HID is a histidine with a hydrogen on the delta nitrogen, HIE has a hydrogen on the epsilon nitrogen, and HIP have hydrogens on both nitrogens —

making it positively charged overall. Since each of these residues have different partial charge states associated with their constituent atoms, it's important we label them correctly. `build_map.py` should automatically look at the structure of the residue and re-label accordingly, but if you want to be extra safe you can check yourself.

`build_map.py` will produce three files: `1dfj_0_sim_map.dx`, `1dfj_0_sim_map.tcl` and `1dfj_0_sim_map.pdb`. The first is our actual STID map, the second is our dipole map used to build the STID map, and the last file is a single pdb frame centred in the density map. We can now take a look at these using VMD again, are they correctly aligned? You can view the tcl map by opening up the tcl console under Extensions in the VMD main window, and typing:

```
source 1dfj_0_sim_map.tcl
```

Now run `build_map.py` on the ligand and we can move on to the docking.

### 3.2.1 Common issues

The main issue you might encounter when running `build_map.py` is that it will not be able to match all the atoms in your system to a corresponding atom in the data file. Your simulation will not run if there are not forcefield parameters available for some of the atoms, so this is not an issue regarding missing charges in the topology file. Any failure notice you get should tell you that it couldn't find the corresponding atomtype from an available list in `biobox/classes/molecule.py`. You'll need to add this atomtype into the metadata list (just a big list that the code can always access) in `molecule.py`. First make a note of what the atomtype is (say, HD3 - a hydrogen). Then go into `biobox/classes` and open up `molecule.py` in a text editor. Scroll down until you find the line that begins with:

```
self.knowledge['atomtype'] = {"C":  "C", "CA":  "C"...
```

Simply expand the end of this list to include information as to the atomtype of HD3 so it becomes:

```
, "HD3" :  "H"}
```

## 3.3 Docking

This is by far the bottleneck in terms of execution time, so if possible we would recommend trying to run this on a supercomputer or at least a powerful PC. There are lots of optional parameters for `dock.py`, but in this tutorial, we will be using most of the default values. We can begin our docking by typing:

```
dock.py -ir 1fdj_0_sim_map -il 1dfj_1_sim_map -np 12
```

Similar to our previous command flags, `-ir` gives the name of the receptor, `-il` the name of the ligand, and `-np` the number of processors to run in parallel. Notice that for both the receptor and ligand we omit any file extension. This is because `dock.py` loads in both the pdb and the dx file, so it only requires the bit beforehand. This also means that you need both the dx and corresponding pdb to have the same name, which is the default from `build_map.py`. You should soon see a message providing the first step's calculated energy (after some preamble about loading in your files). If it's taking a few minutes to start up, it might be worthwhile running elsewhere. It reports a step's energy 900 times in total, so if it takes 10 minutes to report on the energy you're looking at a week-long docking procedure.

`dock.py` will create a folder called `models/` to place any found structures as well as the starting structures it uses. If `models/` already exists, then the old folder will be renamed with the time and date (e.g. `models_old_1524_25_06_2019` following the `TIME_DAY_MONTH_YEAR` format) and a new `models/` folder created. 72000 evaluations are made in total, and depending on how many of them are good, JabberDock could generate thousands of poses. JabberDock clusters them (based on the similarity of the applied rotation and translations) into 300 models that are returned to you in the `models/` folder. If you want to change the number of models generated, you can raise the flag `-ns`, for instance `-ns 500` returns 500 clustered poses. Note that by making this number too low you will likely be omitting some useful poses, as the score is not used to determine what is clustered.

The initial structures are usually labelled as `initial_1dfj_0_sim_map.dx` and so on. These are just your input structures translated to the origin so that both the ligand and receptor have the same starting point, and the space in which we roto-translate the ligand is a bit smaller (to save on computation time).

For each returned docked pose, there are three types of files: a `jabmodel_x.pdb`, `model_x.dx` and `model_x.pdb`. `x` denotes the model number, with each set of three corresponding to one docked pose. The combination of the `initial_1dfj_0_sim_map` model with each model `x` is the predicted total complex. The `jabmodel_x.pdb` is the STID map's isosurface in pdb form and is useful if you want to perform further calculations using the map without loading in the entire density. `model_x.dx` is our overall density roto-translated, and `model_x.pdb` is the actual structure we're interested in, as it's the ligand's coordinates we suspect is a potential docked position paired with `initial_1dfj_0_sim_map.pdb`.

The last file in the folder is `model_solutions.dat`. This contains, respectively, the necessary translations ($x$, $y$, $z$), rotation axis ($I_x$, $I_y$, $I_z$) and angle (deg) required to recreate our models from `initial_1dfj_1_sim_map.pdb`. The penultimate number on each row is the corresponding score, which `rank.py` uses to rank the structures. The final number corresponds to the number of structures that were geometrically clustered to give that one docked pose, so you'll probably notice that results with a higher score will likely have a few more models associated with them. Each row number corresponds to the respective model file, so the very first row will correspond to `model_0.pdb` and so on.

If your simulation fails at some point or there is a timeout, you can use the restart function to continue from a checkpoint file. By setting `-r 1` as a flag, you will be able to continue from a checkpoint file if one is available (i.e. the simulation didn't crash too soon).

## 3.4 Ranking

Finally, we can rank our solutions so that we know which model numbers are our top-performing predictions. First move into the `models/` folder:

```
cd models/
```

Then type:

```
rank.py
```

This produces a file called `ranked_scores.dat`. There will be one row for each docked pose, each row containing the score of the docked pose followed by its model number. The file is ordered in terms of score, so the first row will correspond to what JabberDock determined to be the highest scoring result. Have a look at the top few results. Do a few look like the structure of the 1DFJ structure you can find on the pdb database? Hopefully!

If you've followed this tutorial with a different set of structures then keep in mind that the first few ranked results won't necessarily correspond to your actual docked complex (there will be a lot of false positives). Our benchmark showed we were only successful in 54% of cases — i.e. we had at least one good prediction in the top 10 results. If you have any, this would be a good time to filter the results by any experimental information you have such as residue contact distances.

We hope this tutorial has been helpful. We are always open to suggestions/constructive criticisms to improve, and if you've encountered any difficulties or bugs then please email at l.s.rudden@durham.ac.uk. If JabberDock has been useful in any capacity, then please cite the following publication:

https://pubs.acs.org/doi/10.1021/acs.jctc.9b00474.

We wish you all the best in your work!

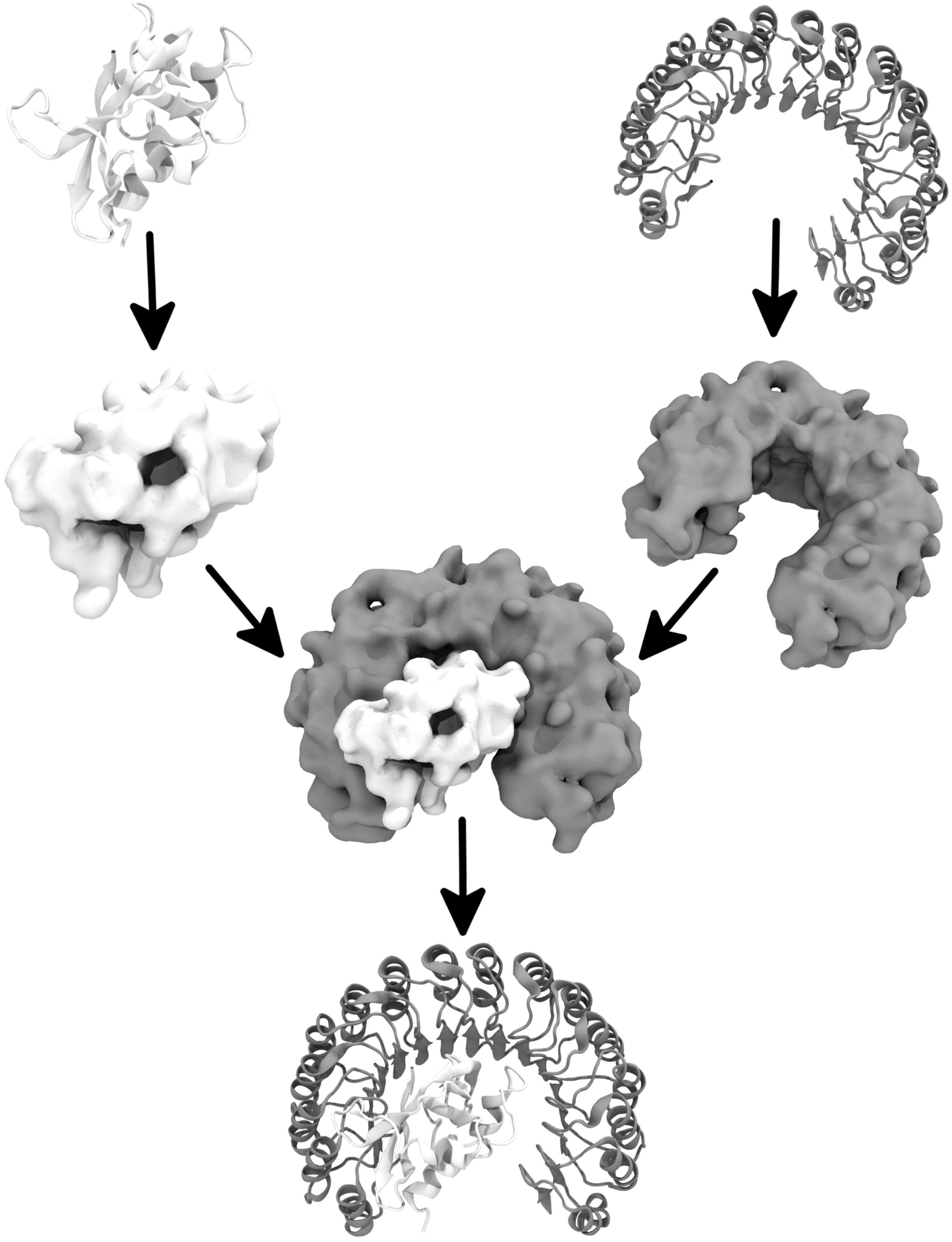


**Figure 4:** *Our final docked result!*