JabberDock 1.0 User Manual

Protein docking using a single descriptor for
protein surface, electrostatics and local dynamics

Lucas S.P. Rudden

CHAPTER

ONE

INTRODUCTION

1.1 What is JabberDock?

JabberDock is a blind protein-protein docking alogirthm that takes advantage of our unique volumetric representation which we call Spatial and Temporal Influence Density (STID) maps. It simultaneously accounts for a protein's shape, electrostatics and local dynamics. By encompassing these factors within a singular representation, we bypass the impact of significant electrostatic changes that can occur due to poor side chain packing. This is typically ignored due to the lack of time-dependent information. In addition, our method minimises the number of custom weightings used in the scoring function. In the original work, we show that it has, on average, a 55% success rate in line with the CAPRI guidelines for the entire benchmark. We show that we can resolve complexes, or at least gain significant insight to the binding site, regardless of flexibility.

Our time-dependency element requires the output from a molecular dynamics (MD) production cycle. We used Gromacs in our approach as it is fast and relatively easy to use. Since only ~500 ps of MD production is actually required, coupled with a CUDA compiled version of gromacs, a run (minimisation to production) would typically only take 20 - 30 minutes. The details of the simulations for the benchmark are outlined in the paper, including our chosen force field. However, in principle any MD engine could be used in combination with any thermodynamically reliable force field.

STID maps are thus built using a multi-PDB trajectory, and requires our in house python tool BioBox. This step is usually quick, taking roughly 5 minutes. Finally, the resultant maps are fed into the docking algorithm.

JabberDock makes use of the particle swarm optimisation "kick and reseed" technique, to efficiently navigate the energy landscape and return candidate models that are geometrically isolated. The scoring function used to create the potential energy surface is based on a surface complementarity score between isosurfaces derived from the volumetric representation. Once a sample set of candidate models are returned, an additional method that looks at the alignment of charge at the interface using the dipole maps re-ranks the solutions.

Within the JabberDock folder, will notice the auto_scripts folder, and the methods folder.

auto_scripts contains all the command line arguments one can make, and is the primary focus of this manual. Inside the methods folder, you will find a series of classes and functions that JabberDock uses during its calculations. Many of the functions avaliable here will likely also be of interest to you, with the ability to roto-translate the maps produced by JabberDock contained within this folder, as well as many other geometric operations not necesserily related to JabberDock's protocol. It includes a function that converts forcefield information from a .rtp file into the .dat files JabberDock uses to generate its maps. This latter function is discussed in more detail below in the JabberDOck Module documentation. When you have a spare moment, it is worth looking through these python files and the associated documentation, as some may be of use to you in your work if you find the docking algorithm of help.

1.2 Reading & using this manual

The process of taking two individual PDB files from the PDB database and returning a predicted complex has several critical steps to it that can seem fairly complex. In this manual, we outline the different steps and levels of autonomy that can be applied; from simply inputting two PDB files and receiving a set of predicted complexes, to tuning specific stages. There are default parameters for each step, so if a user wishes to control only one aspect of the process but let the rest be entirely autonomous that is also possible.

After a brief installation guide, the command to run everything in default is described first, so a user who does not require any further detail can simply use this first section. Following this, each primary feature is described at least once in chronological order, and examples are given for commands that can be submitted. The specific calls made to scripts are described within each process, as well as the details behind each script, should a user wish to have a greater degree of control.

CHAPTER

TWO

INSTALLATION

In order to install/run JabberDock, you should have the following programs installed:

- Gromacs v5.0 or greater
- VMD v1.9.3
- BioBox (and the accompanying python packages)
- The python package cython

JabberDock comes with the functionality to use a CUDA compiled version of gromacs. If you wish to use CUDA functionality then please specifiy while using gmx_run. Note that if you are going to be using GPU accelerated gromacs then you must have a graphics card installed. Additionally, JabberDock will always default to GPU ID 0 (unless specified otherwise), which if you only have one card mounted will be the same as that providing your display. To avoid issues, we recommend that you install the CUDA functionality only if the machine uses no display, or you have more than one graphics card on board. Alternatively, if you want to do the MD steps yourself then you won't need gromacs to be installed at all.

The script install.sh, present in the JabberDock folder, contains the default paths and settings. It is this file you will need to alter if you wish to install elsewhere, or if you want to alter any of the specific settings JabberDock uses, such as the GPU ID. The default parameters used are given in a table for each command that provide the possible flags. Alternatively, you can view the options and defaults interactively by using the command followed by -h (e.g. jabberdock.py -h).

CHAPTER

THREE

COMMANDS

3.1 jabberdock

Create a new folder where you want your workspace to be and place the two PDB you wish to get a prediction from in here. Then, on the command line, type:

jabberdock.py -i1 pdb1 -i2 pdb2

Where pdb1 and pdb2 are your two pdb input files. Note that whilst JabberDock and BioBox will do their best to navigate difficult pdb files (e.g. with charges, ligands etc.)., it is not foolproof, and it is worth checking over your pdb file first to ensure that it is sensible. The forcefield used can also be specified here, though the default is amber03.

The jabberdock command does offer some further flexibility, albeit it is limited. The following provides an example input command using all the possible flags:

jabberdock.py -i1 receptor.pdb -i2 ligand.pdb -ff /home/lucas/amber14sb.ff -np 12 -ntomp 4

-gpu 1111 -ff_dat /home/lucas/amber14sb.dat

We use ff to indicate the forcefield of our choice, you can alternativly use just the names of the forcefield folders in the gromacs/top/ folder. We specify the number of processors we want to use with np, and the number of OpenMP threads with ntomp. More information on this is avaliable in the Gromacs documentation. Note that specifying these flags will indicate to JabberDock that you want to use gmx_mpi, so please ensure you have the correct version of gromacs compiled.

GPU functionality can be indicated by the gpu flag, with the gpu ids we want to map to following (for instance, you could also use 1122 etc.). Again, please refer to the Gromacs website for more information on this. Finally we've specified the data file that BioBox will use to generate our STID maps via ff_dat. This contains the charge information associated with each atomtype. JabberDock contains a function in ... that can convert a ... file (nominally within the topology files gromacs provides) into the necessary data file that BioBox can read.

Table 1: *Possible flag usage with the jabberdock command*

Flag	Input	Default	Required?	Description
i1	receptor.pdb	N/A	Yes	Receptor PDB file
i2	ligand.pdb	N/A	Yes	Ligand PDB file
ff	Forcefield	amber03	No	Forcefield name. Note that amber03 defaults to the version held in Gromacs. You can use similar naming schemes for other default forcefields, but if you are using your own forcefield you'll need to include the full name (including directory location).
np	Number of processors	1	No	The number of processes to run in parallel. This applies to both the MD and the docking phase later on. The default runs both in serial, so if specifying more please ensure you have the correct version of gromacs installed.
gpu	GPU IDs to match with (MD)	0000	No	The gpu IDs to map to during the MD. (see the gromacs website on GPU usage for more information). This requires a CUDA compiled version of gromacs to operate. If not specifed, then GPUs are not used, but if the flag is specified then JabberDock will map to your first GPU. Be wary, this can cause issues if you are simultaneously using your GPU for anything else, for instance rendering your desktop.
ntomp	OpenMP threads per rank	0	No	The number of OpenMP threads per rank. The zero corresponds to running gromacs in serial.
ff_dat	Forcefield data file	amber03.dat	No	Location of the forcefield data file which contains the charge information associated with your forcefield. Some example ones (including the default amber03.dat) are in biobox/classes. Note that the default assumes that biobox is installed in your home directory.

3.2 gmx_run

The gmx_run commands offers a means to streamline much of what gromacs offers into the producing the necessary simulation files BioBox needs to create the STID maps. The only requirement for build_maps is that there is a multipdb input containing the simulation data of only the protein. This step is avaliable more for those who have limited knowledge of MD and / or MD engines, but want a bit more control than what JabberDock offers.

A command using the default settings would look a bit like the following:

gmx_run.py -i protein.pdb

Where protein is the name of the pdb file you want simulating. gmx_run will produce a multipdb containing the simulated protein. Note that VMD is required for gmx_run to run smoothly, so please ensure it is installed on your machine. The output multipdb will be called protein_sim.pdb.

It is good practice to be always be cautious of any MD simulation data, as a parameterised forcefield will not provide an exact insight into experimentally found behaviour. However, it is outside the scope of this manual to address that matter here, and there is already a great deal of existing literature. The majority of well parameterised forcefields used for protein systems (amber, opls, charmm etc.) are shown to replicate consistant and reliable thermodynamic statistics. It is therefore the users responsibility to check whether their forcefield of choice has been shown to produce reliable results for their system, particularly when using temperatures and pressures the forcefield was not parameterised for.

Table 2: *Possible flag usage with the gmx_run command*

Flag	Input	Default	Required?	Description
i	protein.pdb	N/A	Yes	Input PDB file
ff	forcefield	amber03	No	Forcefield reference name, or directions to location (amber03 is in the top folder in gromacs)
ts	Time Step	2 fs	No	Timestep used for the simulation
ns	Number of steps	300000	No	The number of steps to perform in the simulation. Default is 3000000 steps * 2 fs = 600 ps of data
dt	Dumptime	2500	No	How many steps until Gromacs saves a snapshot. Default is 2500 steps * 2 fs = 5 ps.
t	Temperature of simulation	310.15 K	No	Temperature to run simulation at (both NVT and NPT).
p	Pressure of simulation	1 bar	No	Temperature to run NPT simulation at.
s	Number of frames to skip	None	No	The number of frames to skip when parsing the protein simulation data out of the raw files. The default is none, giving a frame every 5 ps under the default settings. Setting this to two will give a frame every 10 ps etc.
np	Number of processors	1	No	The number of processes to run in parallel. The default is to run gromacs in serial, specifying more processors than 1 will use the gmi_mpi version of gromacs, so please ensure this is compiled correctly.

Table 2 cont.: *Possible flag usage with the gmx_run command*

Flag	Input	Default	Required?	Description
ntomp	Number of OpenMP threads	0	No	Number of OpenMP threads to use if running in parallel. The default is 0 to correspond to running gromacs in serial. Please refer to the gromacs documentation for more information on this, and what's best for your computer's architecture.
gpu	GPU ID	0000	No	GPU ID to map to with the MD. This requires a CUDA compiled version of gromacs. If the flag is not specifed, then JabberDock will assume that you are not using any GPUs. If the flag is specified then JabberDock will map to your first GPU. Be wary, this can cause issues if you are simultaneously using your GPU for anything else, for instance rendering your desktop.
minim	minimisation script	minim.mdp	No	Minimisation script to use for the first step of the MD. There is one avaliable in the auto_scripts folder already which is the default. If you wish to use your own, please provide the location here.

3.3 build_maps

The build_maps command takes a multipdb and converts it directly into a STID map for use in our docking protocol. Whilst the typical run time to generate the maps is fairly short (~2 minutes), the memory requirements can be quite demanding. There is ongoing work on this, for now however, we recommend that users have at least 8 GB of RAM to avoid any issues. A typical command using build_maps using the default settings would look like the following:

build_maps.py -i protein_sim.pdb -ff_dat ~/biobox/classes/amber03.dat

protein_sim.pdb is our multipdb simulation file output by gmx_run (or otherwise). We then need to point to the data file that corresponds to the forcefield used. amber03.dat is avaliable in the biobox/classes/ folder, but you can produce your own one accordingly using the ... function in JabberDock. This is discussed in more detail in ... .

Under the default settings, three files will be output: 1) protein_sim_map.pdb, 2) protein_sim_map.dx, 3) protein_sim_map.tcl. The first file is a snapshot from the halfway through the simulation that sits within the STID map. The second is the STID map used by JabberDock for docking. The last file corresponds to the dipole map, and is produced by default, but can be toggled off. This is necessary if one wants to perform the post processing re-ranking using dipole_rerank.

Table 3: *Possible flag usage with the build_maps command*

Flag	Input	Default	Required?	Description
i	protein_sim.pdb	N/A	Yes	Input simulation multipdb file (e.g. the output from gmx_run)
ff_dat	Forcefield data file	N/A	Yes	Forcefield charge data file that corresponds to the forcefield used in the gromacs MD. The default used in gmx_run is amber03, so there is a amber03.dat file in biobox/classes/ that can be used.
ff	Forcefield	amber03	No	Forcefield name. Note that amber03 defaults to the version held in Gromacs. You can use similar naming schemes for other default forcefields, but if you are using your own forcefield you'll need to include the full name (including directory location).
np	Number of processors	1	No	The number of processes to run in parallel. This applies to both the MD and the docking phase later on. The default runs both in serial, so if specifying more please ensure you have the correct version of gromacs installed.

Table 3 cont.: *Possible flag usage with the build_maps command*

Flag	Input	Default	Required?	Description
gpu	GPU IDs to match with (MD)	0000	No	The gpu IDs to map to during the MD. (see the gromacs website on GPU usage for more information). This requires a CUDA compiled version of gromacs to operate. If not specifed, then GPUs are not used, but if the flag is specified then JabberDock will map to your first GPU. Be wary, this can cause issues if you are simultaneously using your GPU for anything else, for instance rendering your desktop.
ntomp	OpenMP threads per rank	0	No	The number of OpenMP threads per rank. The zero corresponds to running gromacs in serial.
ff_dat	Forcefield data file	amber03.dat	No	Location of the forcefield data file which contains the charge information associated with your forcefield. Some example ones (including the default amber03.dat) are in biobox/classes. Note that the default assumes that biobox is installed in your home directory.

3.4 dock

In this step, we use the POW optimisation engine to generate a series of predictive complexes that are ranked by the JabberDock scoring function. During this step, JabberDock will only use the surface complementarity scoring function to analyse the results. For more information on this, please refer to the background paper.

This is by far the most time consuming and memory demanding step, we therefore recommend running this on a supercomputer if you have the ability to. It is possible to run on a home PC, and for small proteins (<5 kDa), this step can be run within a reasonable timeframe (<2 hours) on a small number of processors (~4). No benchmarking has been performed using only one CPU. While increasing the number of CPUs will indeed reduce the calculation time, the RAM requirements also scale with this. The largest protein complex benchmarked in the literature for JabberDock, 1N2C - at a mass of 286 kDa, required ~65 GB of RAM across 28 CPUs. That was to run in 60 hours, the longest calculation in the benchmark. The supplementary information in the literature provides more information on the scaling of mass and time. So please be aware of what you're asking for when running JabberDock on clusters, and what resources are avaliable. One of the primary goals of JabberDock 2.0 is an increase to the computational speed and efficiency.

Similarly to above, the dock command using solely the default command is given below:

dock.py -ir receptor_sim_map -il ligand_sim_map

Where receptor_sim_map and ligand_sim_map are the output pdb, dx, tcl filenames from build_maps. Do not use the suffix file names, as these are automatically interpreted by dock. Whilst dock only uses the scoring function, the tcl maps are aligned with the starting structures used to initialise the conformational space by default for use in the dipole_rerank function. These starting structures are called initial_protein_sim_map.pdb and so on. There is an option to turn off the dipole alignment, as explained in table 4, using this will simply not produce the initial version of the tcl files.

dock will create a folder called models within the folder you're currently in and place all the found solutions in there, including the initial structures. If a models folder already exists, then it will rename that old folder to the current date PLUS SOME chronological NUMBER if there are more than one for that date. In addition, a logfile called time_log.dat will be produced which contains the console output from POW. POW will narrow down the number of found solutions into how many geometrically isolated solutions you request (the default is 300) using a kmeans clustering algorithm. It is possible using the parse.py script found in POW, along with an altered input_ensemble script, to change the number of independent predictions finally produced after POW has finished, so you could instead request 600 geometrically isolated cases from however many structures POW found. This is because POW keeps a record of all the solutions in another data file called filtered_log.dat. For this, you simply need to alter the number in front of no_samples in the input_ensemble script, and call:

python /user/home/POW/parse.py input_ensemble

The location is just an example. Depending on the number of solutions requested, say the default 300 requested, you will find 900 (so three times more) the number of output structures. These will include the ligand pdb file moved into location, the STID map dx file, and a jabmodel pdb file, which contains just the isosurface generated from the STID map in PDB format.

The first is labelled as model_0.pdb, model_0.dx, and jabmodel_0.pdb respectively, with the number increasing for each solution (up to 299 in this case). When the results have been ranked, these numbers will correspond to the last column.

The output file containing the scores, the default is model_solutions.dat, contains 9 columns. The first 3 gives the translation vector to move the ligand from its initial position on the origin (the origin ligand is initial_ligand_sim_map.pdb), columns 4-6 define the rotation axis, column 7 gives the angle about this axis that the protein was rotated. Column 8 gives the scores for each model described by those roto-translational vectors, the final column indicates the number of found solution structures that were grouped geometrically to give the one model you see in the row. The row number corresponds to the model number in the models/ folder.

Table 4: *Possible flag usage with the dock command*

Flag	Input	Default	Required?	Description
ir	Receptor name	N/A	Yes	Name of the receptor input file prefix shared between the pdb, dx and tcl output from build_maps
il	Ligand name	N/A	Yes	Name of the ligand input file prefix shared between the pdb, dx and tcl output from build_maps
iso	isovalue cutoff	0.43	No	Isovalue to generate the isosurface from the STID map to calculate the surface complementarity scoring function
d	distance cutoff	1.6	No	Distance cutoff to consider valid contact between points on receptor and ligand isosurfaces
l	logfile	pow_log.dat	No	Logfile that keeps track of all geometric arrangements and corresponding scores. This is a necessary file when running a restart, and is distinct from filtered_log.dat which contains all found solutions.
ns	Number of Samples	300	No	Number of geometrically isolated solutions to be reduced from the number of predictions deemed acceptable by JabberDock. If you are concerned about losing possible predictions, then it is recommended to increase this number.
m	Space Multiplier	1.5	No	Multiplier to the conformational space the ligand is allowed to move around in. The receptor's size is used as a foundation, so if 2 is used, that's twice the size of the receptor. Since both are centered at the origin, this shouldn't really be necessary - but is included in case you have a funny shaped receptor / ligand. Note that increasing this number will make it harder to generate solutions, as POW will spend longer placing the ligand in sites that are not in contact with the receptor.

Table 4 cont.: *Possible flag usage with the dock command*

Flag	Input	Default	Required?	Description
b	buffer	0 Ang.	No	Additional space to include alongside the geometric space of the receptor when roto-translating the ligand. Units are in angstroms. This is added after the multiplier step (see above).
a	Angle	180°	No	The angle to twist the ligand around in space about some axis through the center of the molecule. The angle runs from -a to a, so 180° indicates the entire rotational space of the molecule. Unless you know what orientation the ligand or receptor are to one another, then it is better to leave this.
n	Name	model_solutions.dat	No	Name of .dat output file which contains the roto-translations and scores. model_solutions.dat is the default filename to be read in by any following commands, so keep that in mind if you use a different name
r	Restart	0	No	If this is a restart run (i.e. continuing from a previous point), then set -r 1. Elsewise, 0 indicates that you are starting from the run from scratch
np	Number of Processes	1	No	How many processes to use if running in MPI, the default is 1 (i.e. serial).
tcl	Dipole Map	1	No	Whether you want the tcl dipole map superimposed onto the initial starting points for the dx and pdb maps, this is necessary for any post-processing. The tcl file should have the same name as your STID map's pdb and dx file, e.g. receptor.tcl. The default is 1 (i.e. True), otherwise set -tcl 0.

3.5 dipole_rerank

dipole_rerank takes the solutions output by dock, and ranks each according to an additional dipole re-ranking procedure that looks at the alignment of charge at the interface using the dipole map. This is a work in progress command, and whilst we mention in the governing literature that we observe an improvement of 12% in the benchmark cases, this is not a universal improvement. Therefore, we would recommend that you don't use this command until we can show, with a specific set of parameters and set up, that this does provide an improvement. We list the instructions here anyway for completeness. An example for the dipole_rerank command is as follows:

dipole_rerank.py -ir initial_receptor_sim_map.tcl -il initial_ligand_sim_map.tcl

-ilpdb initial_ligand_sim_map.pdb

These are all the mandatory arguments. A full list of possible arguments is given below in Table 5. The output re-ranked scoring file will contain one column, the re-ranked scores. These will always correspond to the list given in the solutions data file output by dock.

Table 5: *Possible flag usage with the dipole_rerank command*

Flag	Input	Default	Required?	Description
ir	Dipole receptor name	N/A	Yes	Initial receptor dipole map tcl file output by dock into the models/ folder. The name will be consistant with the dx and pdb files. This is the default behaviour by dock.
il	Dipole ligand name	N/A	Yes	Initial ligand dipole map tcl file.
ilpdb	PDB ligand name	N/A	Yes	Initial PDB ligand file that corresponds to the ligand tcl file specified by -il. This is also output by dock during initialisation, and will share a similar name to the ligand tcl file.
d	Scores filename	model_solutions.dat	No	Data file that contains the roto-translations and scores of the found solutions. This is model_solutions.dat file that would be output by dock under default settings.
o	Output filename	new_scores.dat	No	Name of the output file that contains the new scores.
w	Weighting	1	No	The weighting that is applied to the surface complementarity score if no dipole score if found for a specific model. When a score is found, the two are summed linearly. Set -w 0.0 if you only want contributions from the dipole, though note that this may cause errors in cases where a dipole score cannot be found / calculated due to the lack of coherent, proximal dipoles.

3.6 rank

rank is the final major command possible with JabberDock. It produces a file containing two columns, which are ranked from highest to lowest score. The first column contains the scores themselves, the second the corresponding model number. So, if the first row's second number is 99, then model_99.pdb in the models/ folder is the highest scored protein. It is likely that a correctly predicted pose will be contained within the top 10 solutions. An example using the rank command is as follows:

rank.py

There are default settings for all arguments with rank, so no specific inputs are required to use it unless you've changed any output filenames. By default, rank uses model_solutions.dat as its input file. Any 9-columned file will work provided the columns are in the order specified in dock. If you specify a single columned file (i.e. new_scores.dat output by dipole_rerank), then rank will correctly interpret that. Using any other structured file will result in a failure. The potential arguments for rank are given in Table 6.

Table 6: *Possible flag usage with the rank command*

Flag	Input	Default	Required?	Description
d	Input file	model_solutions.dat	No	Data file containing the scores for each model. Can be either a single or 9-columned file (corresponding to dipole_rerank and dock outputs accordingly). The default corresponds to the dock, surface complementarity output.
o	Output file	ranked_scores.dat	No	Name of the output file containing the ranked scores (from highest to lowest). The file will contain two columns: the first indicates the score, the second the corresponding model number.

CHAPTER

FOUR

THE JABBERDOCK MODULE

All the command line arguments listed above use python commands as provided by either the BioBox, JabberDock or POW modules. If you wish to have an even greater control over how the docking procedure works, how the STID maps are written etc., then the following provides detailed documentation on all the python functions provided by the JabberDock module.

JabberDock.methods.data.**create_pqr_dat**(*in_file*, *out_file*, *template="../biobox/classes/amber14sb.dat"*)

Convert an rtp file (e.g. in gromacs top folder) into an appropiate pqr format that can be interpreted by biobox to form the STID maps.

Parameters

- **in_file** – Name of input rtp file, e.g. aminoacids.rtp in amber03.ff/ folder
- **out_file** – Name of output file that can be read by the pqr interpreter (import_pqr in biobox.Molecule)
- **template** – Template of pqr file. We recommend just using the amber14sb.dat shipped with biobox

JabberDock.methods.geometry.**define_isosurface**(*dx*, *ndarray COM=np.array((0., 0., 0.))*, *float cutoff=0.5*, *ndarray R=np.identity(3)*)

Obtain an isosurface of a density object that contains the vertices, normals to said vertices, faces and the max value of the triangle used to define the isosurface using the marching cubes algorithm. The vertices returned are 3D coordinates. This is defunct now, as when defining a jabber object this is performed automatically, and the isosurface is an associated property of the jabber object.

Parameters

- **dx** – Density map previously imported from _dx format
- **COM** – Centre of mass of the density map. Can be the same as get_center() of corresponding PDB structure.
- **cutoff** – Isolevel cutoff for map
- **R** – Rotation matrix used to define rotation (can be output from rotate_map)

Returns Four arrays - verts, faces, normals and values. Verts contains the vertices of the triangular mesh that makes up the isosurface (V, 3) is the shape, which match the order of the input volume. V corresponds to the unique vertices. Faces contains the triangular faces that refer to verts, (F, 3) is the shape - where F is the number of faces. Normals are the normal direction to each vertex with shape (V, 3). The values contains a measure for the maximum value of the STID quantity nead each vertex (useful for visualisation), with shape (V).

JabberDock.methods.geometry.**distance_list**(*points1*, *points2*, *float cutoff=1.5*)

Get a list of indicies for the vertices that are closest to one another, and use a cutoff to remove points that we're not interested in (i.e. beyond a typical non-bonded cutoff distance) This function has the possibility of using the gpu pairwaise distance function (if uncommented) - though this is unstable for large proteins)

Parameters

- **points1** – First set of coordinates, with shape A x 3

- **points2** – Second set of coordinates, with shape B x 3
- **cutoff** – A cutoff to exclude interactions from

Returns Numpy array containing two arrays - the first contains the distances to the closest point on points2 from points1 for each point on points1 where said distance is less than the cutoff. The second contains the minimum distances from points1 to points2 for each point in points2 where that distance is less than the cutoff. Note that this means that the two don't have to have the same shape.

Returns Numpy array containing two arrays. The first has shape A, the second shape B. Each element in an array contains a boolean referencing whether each point is able to make contact with a point on the other surface within the cutoff.

Returns Numpy array containing two arrays. The first has shape A, the second shape B. Each element in an array references the index to an element in the other array to which the point in the first array is closest to.

Example Say you have two sets of coordinates, A and B, with shape 3 x 3 and 6 x 3 (3 corresponding to x, y, z). Plugging them into here you might get the following arrays:

dist = array([array([1.09]), array([0.58, 1.47])]).

bool_index = array([array([True, False, False]), array([True, True, False, False, False, False])]).

min_index = array([array([4, 4, 0]), array([2, 2, 2, 0, 0, 2])])

dist[0] says that there is only 1 point in A within the cutoff of B. It has a distance 1.09 to that point. dist[1] shows that there are two points in contact with A.

bool_index[0] essentially gives us the index on A to which dist[0] applies to. So only the first point is in range of B, equally with B only the first two points are in range with A.

Finally, min_index[0] provides us with the elements on B that each point is closest to. So the first point in A is closest to point 4 (or index 5) on B, so is point 2. Point 3 on A is closest to the first point on B.

This information together gives us each set of points that satisfy our cutoff, and the distance between them.

`JabberDock.methods.geometry.`**`fast_argmin_axis_0`**`(`*ndarray a*`)`

A function that basically takes the place of numpy.argmin(), but is much faster

Parameters **a** – The array that we need the index for the minimum value

Returns An array of indicies into the array (with len(flattened a))

`JabberDock.methods.geometry.`**`get_center_distance`**`(`*ndarray points*`)`

Return the distance (in angstroems) from a set of points from the center of a molecular surface

Parameters **points** – Input coordinates of molecular surface (n by 3 in shape), where n is the number of points

Returns A Cartesian vector containing the distance of between the series of points input and the geometric centre of the STID map.

JabberDock.methods.geometry.**get_minmax_crd**(*fname_rec*, *fname_times=2*, *fname_buff=0.0*)

Obtain the min max coordinates from a receptors conformational space to generate sensible structures for ensembles in JabberDock. We move the ligand around this space to represent the size the receptor occupies, but we do want it to be slightly bigger.

Parameters

- **fname_rec** – PDB file name of receptor
- **fname_times** – Number greater than 1 to represent extra space added to search space (default is 2, so twice as big as the receptor)
- **fname_buff** – Buffer region at the edges to include (default is 0.0 ang.)

Returns numpy array size 3, containing the x, y and z lengths that provide the sample space about which the ligand will move in.

JabberDock.methods.geometry.**get_mol_center**(*ndarray points*)

Get the geometrix center of a molecular surface, as defined by the vertix coordinates, primarily an internal function

Parameters **points** – Input coordinates of molecular surface (in n by 3 shape), where n is the number of points

Returns A Cartesian vector containing the coordinates of the geometric centre.

JabberDock.methods.geometry.**redefine_coord**(*dx*, *float cutoff=0.5*)

Define a coordinate system given a cutoff for the isosurface based on the input information of the dx map. The function is also avaliable to rotate / translate maps as necessary, providing a map of points at a spectific isovalue cutoff. The rotation already occurs given a specific delta in the input map.

Parameters

- **dx** – Density map previously imported from dx format (i.e. bb.Density() object)
- **cutoff** – Isolevel cutoff for map

Returns An array which contains the coordinates of every voxel point greater than the isovalue cutoff given

JabberDock.methods.geometry.**rotate_map**(*dx*, *ndarray COM=np.array((0., 0., 0.))*, *float angle=0.*, *float axisx=0.*, *float axisy=0.*, *float axisz=0.*, *R=np.identity(3)*)

Method to rotate electron density maps about the an axis of rotation from the origin in place

Parameters

- **dx** – Density map previously imported from dx format (i.e. bb.Density() object)
- **COM** – Centre of Mass of corresponding PDB (if using PDB as basis for roto-translations)
- **angle** – Angle to shift by (must be in deg)
- **axisx** – Point in x for the axis of rotation
- **axisy** – Point in y for the axis of rotation
- **axisz** – Point in z for the axis of rotation
- **R** – Rotation matrix if already defined

Returns The rotation matrix used to rotate the map

JabberDock.methods.geometry.**rotate_pdb**(*pdb*, *float angle=0.*, *float axisx=0.*, *float axisy=0.*, *float axisz=0.*, *R=np.identity(3)*)

Method to rotate pdb structures about an axis of rotation from the origin in place

Parameters

- **pdb** – PDB structure (i.e. bb.Molecule() object)
- **angle** – Scalar for angle to rotate (units of degrees)
- **axisx** – Point in x for the axis of rotation
- **axisy** – Point in y for the axis of rotation
- **axisz** – Point in z for the axis of rotation
- **R** – Rotation matrix if this function has already been applied

Returns The rotation matrix used to rotate the protein

JabberDock.methods.geometry.**translate_map**(*dx*, *float x=0.*, *float y=0.*, *float z=0.*)

Method to translate electron density maps based on origin point of grid in place

Parameters

- **dx** – Density map previously imported from dx format (i.e. bb.Density() object)
- **x** – x coordianate to shift by (units in Ang.)
- **y** – y coordianate to shift by
- **z** – z coordianate to shift by

JabberDock.methods.geometry.**translate_pdb**(*pdb*, *float x=0.*, *float y=0.*, *float z=0.*)

translate a whole PDB structure by a given amount in place. Function is defunct relative to biobox.molecule.translate()

Parameters

- **pdb** – PDB structure (i.e. bb.Molecule() object)
- **x** – The desired x translation amounts
- **y** – The desired y translation amounts

- **z** – The desired z translation amounts

JabberDock.methods.geometry.**vdw_energy**(*ndarray m1*, *ndarray m2*) → float

Calculate the vdw energy between two PDB structures (typically bb.Molecule()) - a check for clashes

Parameters

- **m1** – Coordinates for structure 1
- **m2** – Coordinates for structure 2

Returns The calculated VdW scalar energy

class JabberDock.methods.jabber.**Jabber**

A Jabber object contains information pertaining to an isometric surface that described an input density at a desired threashold - in the literature it is known as the Spatial and Temporal Influence Density (STID) map

It contains information on the vertices, normals to said vertices, faces and the max value of the triangle used to define the isosurface using the marching cubes algorithm. The vertices returned are 3D coordinates

distance_list(*self*, *jabber2*, *float cutoff=1.6*)

Get a list of indicies for the vertices that are closest to one another, and use a cutoff to remove points that we're not interested in (i.e. beyond a typical non-bonded cutoff distance)

Parameters

- **jabber2** – A Jabber() object to get a distance list between (we're interested in jabber.verts here)
- **cutoff** – A cutoff to exclude interactions from - typically between 1 Ang. and 3 Ang. (non-bonded cutoff is what we're interested in)

Returns Numpy array containing two arrays - the first contains the distances to the closest point on points2 from points1 for each point on points1 where said distance is less than the cutoff. The second contains the minimum distances from points1 to points2 for each point in points2 where that distance is less than the cutoff. Note that this means that the two don't have to have the same shape.

Returns Numpy array containing two arrays. The first has shape A, the second shape B. Each element in an array contains a boolean referencing whether each point is able to make contact with a point on the other surface within the cutoff.

Returns Numpy array containing two arrays. The first has shape A, the second shape B. Each element in an array references the index to an element in the other array to which the point in the first array is closest to.

Example Say you have two sets of jabber objects, jabber_A and jabber_B, with jabber_A.verts and jabber_B.verts with shape 3 x 3 and 6 x 3 respectively (3 corresponding to x, y, z). Plugging them into here you might get the following arrays:

dist = array([array([1.09]), array([0.58, 1.47])]).

bool_index = array([array([True, False, False]), array([True, True, False, False, False, False])]).

min_index = array([array([4, 4, 0]), array([2, 2, 2, 0, 0, 2])])

dist[0] says that there is only 1 point in A within the cutoff of B. It has a distance 1.09 to that point. dist[1] shows that there are two points in contact with A.

bool_index[0] essentially gives us the index on A to which dist[0] applies to. So only the first point is in range of B, equally with B only the first two points are in range with A.

Finally, min_index[0] provides us with the elements on B that each point is closest to. So the first point in A is closest to point 4 (or index 5) on B, so is point 2. Point 3 on A is closest to the first point on B.

This information together gives us each set of points that satisfy our cutoff, and the distance between them.

rotate(*self*, *ndarray COM=np.array((0., 0., 0.))*, *ndarray R=np.identity(3)*)

Rotate a jabber object (STID map) via a rotation matrix in place (presumably obtained through geometry.translate_pdb or geometry.translate_map)

Parameters

- **COM** – Centre of Mass of system
- **R** – Rotation matrix

scoring_function(*self*, *jabber2*, *ndarray dist*, *ndarray bool_index*, *ndarray index*, *float weight=0.5*)

Define a scoring function to obtain a value to obtain the 'goodness of a fit', or surface complementarity, between two isosurfaces at a specific region of the surface. This relies on you first obtaining the set of nearest neighbours within a sensible cutoff with distance_list.

Parameters

- **jabber2** – Second jabber object (STID map) which we're treating as the ligand
- **dist** – distance list containing two lists with distances to their nearest neighbours for proteins 1 and 2 respectively in the cutoff region. Can be created with jabber.distance_list()
- **bool_index** – Series of booleans list (for proteins 1 and 2) which are true when we have an index for an atom that satisfies the cutoff - we need this for access to normals etc. Can be created with jabber.distance_list()
- **index** – Minimum indicies for atoms that are within the cutoff distance (corresponds to the distance lists). Can be created with jabber.distance_list()

- **weight** – An arbiturary weighting, can be used to modify the distance weighting in the scoring function, and tends to give worse scores for less well fitting models. Taken from Lawrence & Colman 1993

Returns A singular score that judges the fit of the two STID maps. The larger, the better

translate(*self*, *ndarray trans=np.array((0., 0., 0.))*)
Translate a jabber object (STID map) via a x, y, z vector in place

Parameters **trans** – Transformation vector

write_jabber(*self*, *fname*)
Write the STID map object so it can be visualised. (Is written as a PDB file with each triangular vertex corner defined as an atom)

Parameters **fname** – Name of output file (will be a .pdb file)

A test function to try and build an additional scoring term using the dipole maps

class JabberDock.methods.dipole.**Dipole**(*d=array([], shape=(1, 0), dtype=float64)*, *c=array([], shape=(1, 0), dtype=float64)*)
A Dipole object contains an ensemble of points in 3D space with an associated dipole value.

combine_dipole(*dipole2*)
Combine two dipole maps, with vectors closer than 1 Ang adding, though scaled by distance

Parameters **dipole2** – Second Dipole() structure

Returns A new dipole map with the combined properties of the two inputs.

dipole_score(*dipole_map2*, *cutoff_low=0.0*, *cutoff_high=2.0*)
Return a score based on the alignment of the dipoles

Parameters **dipole_map2** – The Dipole map (in jd.Dipole() form) that is being docked into self.

Returns An overall score for the dipole alignment.

dist_cutoff_list(*verts*, *cutoff_low=2*, *cutoff_high=5*)
Return a list of distances that are between two cutoffs

Parameters

- **verts1** – coordinate for first protein (the stationary one to be docked into)
- **verts2** – coordinate for second protein (the roto-translated structure)
- **cutoff** – A cutoff to exclude interactions from - typically between 1 Ang. and 3 Ang.

Returns Numpy array containing four separate arrays. The first contains the distances between the two specificed cutoffs for the first set of input dipoles. The second is a boolean array where the length corresponds to the number of dipoles in the first dipole map. Where, True, the corresponding dipole

has a matching dipole on the second map (the ligand). The third and fourth arrays are the same but from the second dipole map's perspective.

`import_dipole`(*dip_name*)
Imports a dipole map written in the tcl format. This can be written by biobox.molecule.get_dipole_map.

Parameters `dip_name` – Name of dipole tcl file

`rotate_dipole`(*COM=array([0., 0., 0.]), angle=0.0, axisx=0.0, axisy=0.0, axisz=0.0, R=array([[1., 0., 0.], [0., 1., 0.], [0., 0., 1.]])*)
Method to rotate electron density maps about the an axis of rotation from the origin in place

Parameters

- `COM` – Centre of Mass of corresponding PDB (if using PDB as basis for roto-translations)
- `angle` – Angle to shift by (must be in deg)
- `axisx` – Point in x for the axis of rotation
- `axisy` – Point in y for the axis of rotation
- `axisz` – Point in z for the axis of rotation

Returns The rotation matrix used to rotate the map

`total_dipole_calc`()
Calculate the total value of all the dipoles summed in a dipole map, a test function to see if this is minimised during binding

Returns An 'energy' corresponding to the overall sum

`translate_dipole`(*trans=array([0., 0., 0.])*)
translate a Dipole map by a given amount in place.

Parameters

- `x` – The desired x translation amounts
- `y` – The desired y translation amounts
- `z` – The desired z translation amounts

`write_dipole`(*fname*, *radius=0.3*)
Write a dipole map in a tcl format that can be read in by VMD

Parameters

- `fname` – Name of output tcl file
- `radius` – Radius of drawn cone if desired (visualisation purposes only)