PyAutoFEP user manual

trunk version

February 11, 2021

Contents

1 Getting started

1.1 About PyAutoFEP

PyAutoFEP is an automation tool to prepare and analyze FEP calculations. It is written in Python3 and relies on rdkit library to work with molecules, networkx to represent perturbation maps, and alchemlyb to analyze FEP results. Currently, Gromacs is the only supported Molecular Dynamics (MD) code, but we plan to add others in a future release.

PyAutoFEP is divided into three modules: a perturbation map generator, a hybrid-topology and system builder, and an analysis script. It allows complete freedom regarding the choice of the λ values for Van der Waals and Coulomb interactions of states A and B. Also, enhanced sampling techniques Hamiltonian Replica Exchange (HREX), HREX with solute tempering (REST), and HREX with solute scaling (REST2) are supported. Overall, we aim to make PyAutoFEP as flexible as possible.

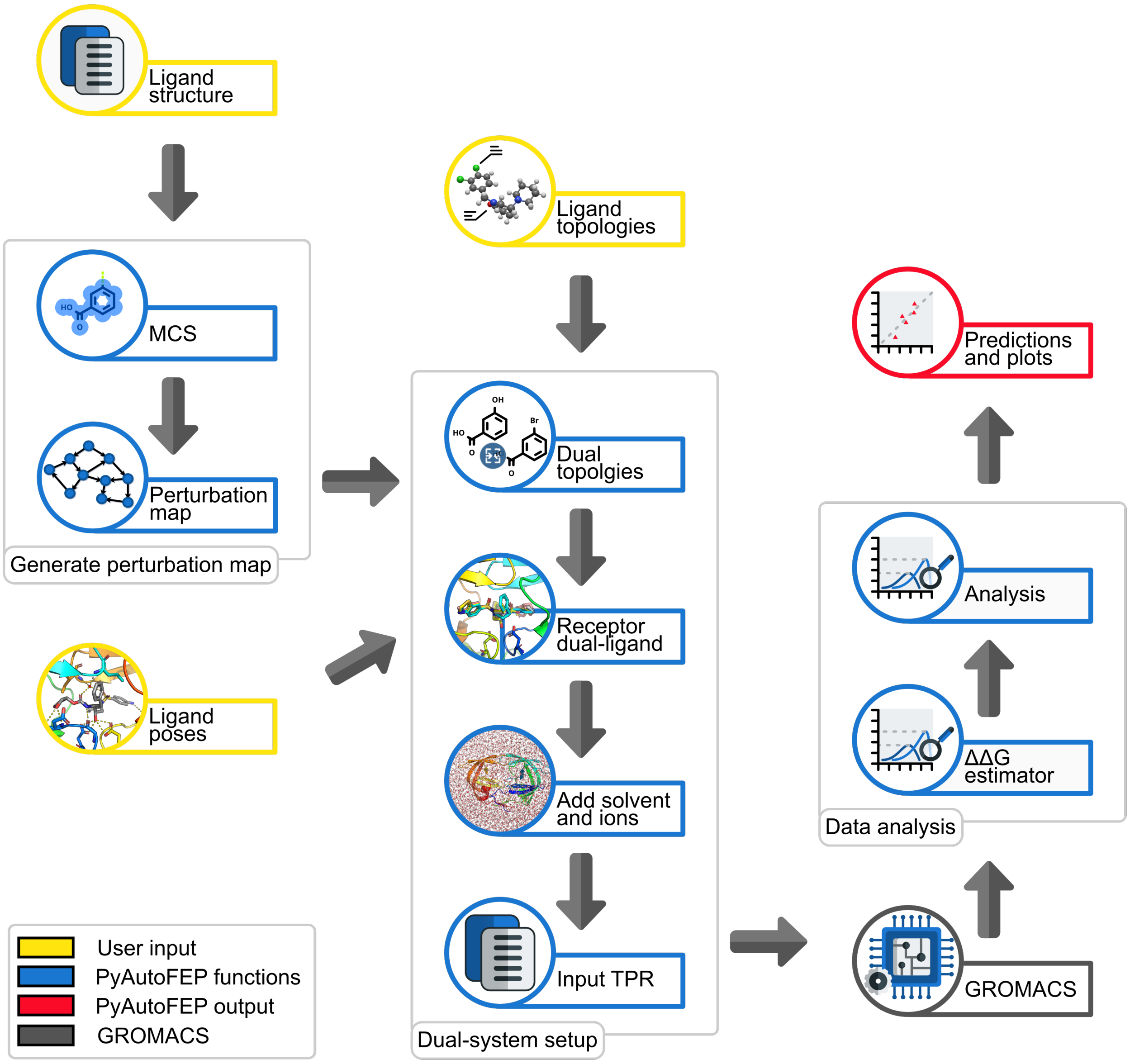


Figure 1: Overall workflow of PyAutoFEP.

PyAutoFEP is open source software and its source code is freely available at [LINK]. Contributions are welcome. Please, report any bugs at [LINK].

1.1.1 Legal notice

1.2 Installation

1.2.1 Requirements

- Common GNU programs: Bash, awk, tar
- Gromacs 2016 or newer
- Python 3.6+
- rdkit 2019.03+
- networkx 2.3
- alchemlyb 0.3 & pymbar 3.0.4
- openbabel 2.4 (sparsely used, mainly to load receptor files in *prepare_perturbation_map.py*. Note that openbabel 3.0+ removes)

Optional. The following are not required to run basic calculations in PyAutoFEP but are needed for users to intend to use specific functions.

- matplotlib (to plot data in *prepare_perturbation_map.py* and *analyze_results.py*)
- biopython (allows sequence alignment when reading initial pose data)
- mdanalysis (allows use of atom selection language in some contexts)
- pytest (required to run Python tests)

1.3 General notes

PyAutoFEP is Linux-compatible and was not tested on Windows or macOS. All text files are supposed to use Unix line-endings (ie: \n), and paths are supposed to use Unix separator (/).

Across this manual, the following conventions are used.

Output message from PyAutoFEP: **[INFO] This is an output message**

Command line argument or configuration file option (they are equivalent, see below): argument

Valid value for an option: **value**

Script: *run_this.py*

Implementation note, relevant for Python users and using PyAutoFEP's functions as an API: *This is not required to regular PyAutoFEP operation.*

Reference to other sections of this document: **subsection 1.3**

1.3.1 Configuration files and command-line arguments

All PyAutoFEP scripts accept a configuration file and command-line options. The exact same options can be supplied by either. The value resolution for the options is: default (internal) configuration file → user configuration file → command-line arguments

So, any option not present in the user configuration file neither in the command-line arguments will have the default value. If an option is present in both the user configuration file and in the arguments, the argument value will take precedence.

Configuration files follow ini format, read trough configparser `https://docs.python.org/3/library/configparser.html`. Lines starting with # or **;** are comments. Every option must be inside a section, declared by [and], for instance, as [**this_section**]. Options are read one per line in the format **option = value** An example of a valid ini file is the following:

```
# This is the configuration section for prepare_dual_topology.py (which is the only section in this file, but the [
section ] is mandatory)
[ prepare_dual_topology ]

# Read ligands and topologies from this folder
input_ligands = lig_data

# Read the macromolecule structure from this file
structure = receptor_data/5q17_processed.pdb

# This is the force field directory, it will be copied to each perturbation dir and used to prepare the
MD systems
extradirs = oplsaam.ff

# Options controlling the core-constrained superimposition
# First select the use of it instead of reading all ligand poses
pose_loader = superimpose

# Use this pose as the reference for the superimposition
poses_reference_pose_superimpose = receptor_data/9mv.pdb

# Name of the output. This will be a self-extracting bash file
perturbations_dir = tutorial

# Sets the path to GROMACS executable in the run node, uncomment and modify if needed.
# gmx_bin_run = /usr/local/bin/gromacs

# Options controlling the output, see manual for more info.
Uncomment as needed
# FEP legs are to be submitted to a slurm scheduler
output_scripttype = slurm
# Run these commands at the beginnig of the jog (useful to load modules, importing libs)
output_runbefore = module load python3; module load cuda
# Fine-tune job resources
output_resources = all_cpus: 24
all_gpus: 2
all_time: 24
# Use a python file instead of a binary during the collect step
output_collecttype = python
```

On all scripts, a configuration file can be passed via –config argument. Note that you cannot use a config option in the configuration file itself. This is the only case where a command-line argument cannot be passed from a configuration file.

When parsing the options or arguments, PyAutoFEP tries to infer types from them. The following formats will be read and converted when parsing data. *PyAutoFEP uses eval to parse arguments or options, so any format Python eval can parse will be understood.*

Table 1: General data types understood by PyAutoFEP

Datatype	**Comments**
Dictionary[1]	A data structure storing key and related values. Keys must be unique. *Some accepted keys types, eg: tuple, frozenset, can only be passed when the native Python3 formatting is used.* ”**:**” are used to separate key and values and ”**;**” or newlines can be used to separate key:value groups. A List value can be used using ”**,**” as a separator. The following formats are accepted: • {“option a”: “value1”, “num_option”: 0.123, 2: “also valid”, “list_option”: [0, 1, “b”]} • option a: value1; num_option: 0.123; 2: also valid; list_option: 0, 1, b • option a: value1 num_option: 0.123 2: also valid list_option: 0, 1, b
List	A data structure storing a sequence of elements. Any format can be stored. *List of lists can only be passed using native Python3 format.* The following formats are accepted: • [“native”, “Python3”, “formatting is always”, True, ”accepted”] • one, 2, 3.0, 4.0, five and more stuff • one two several spaces in one value
String	A String is any stream for characters, including spaces. The String is the fallback type in PyAutoFEP, so if anything cannot be detected as another type, it will be a String. No special formatting is needed. Examples: • my_file_name.dat • name with spaces.txt • some 123 thing
Integer	Regular integer. Note that using . in the formatting will force a float type, eg ”1.0” is not the same as ”1”. Decimal numbers only, no ”**.**” or ”**,**”. Examples: • 0 • 123 • -456 • ~~0856~~ (*invalid*)
Float	A number containing a decimal part. Scientific notation is allowed. The number must have either a decimal part, separated by ”**.**”, or a scientific notation exponent, separated by and ”**e**”. Examples: • 3.1416 • 6.02214e23 • 1.985875e-3 • 0.593 • 123.0 • -123.0

Table 1: General data types understood by PyAutoFEP

Datatype	**Comments**
Boolean[2]	A True/False value. The following keywords are accepted: • True • Yes • On • False • No • Off

[1] In most of the places where a dictionary is accepted, a text file containing the same any of the dict structures can also be used. Note that the third allowed format requires a line end between option lines.

[2] Python3 will interpret 0, 0.0, None or an empty string, list or dict as False and anything else as True, but this can lead to problems. Using one of True, On, Yes, False, No, or Off is safer.

All PyAutoFEP scripts accept the following set of global options/arguments:

Table 2: Global PyAutoFEP options accepted by all scripts

Option or argument	**Accepted values**	**Comment**	**Default**
threads	Integer, $x \geq \mathbf{0}$	When possible, use this many threads. **0**: use maximum available.	**0**
progress_file	File name	Use this progress file to save and load data during the run and between runs. See [1.5.3] for further reference.	*Use a hidden file*
mcs_type	**graph** or **3d**	Select MCS algorithm. **graph** is suitable for molecules in which no sterocenters are inverted on perturbations. **3d** is a spatial-guided MCS, which works on perturbed stereocenters but may break rings. See **subsubsection 1.3.4** for further reference.	**graph**
config_file	File name	Read this configuration file. See **subsubsection 1.3.1** (ie: this very section) for further reference	*Do not read a config file*
plot	Boolean	Where possible, plot data using matplotlib.	**Off**
no_checks	Boolean	Ignore all checks and try to go on.	**Off**
verbose	Integer or repetition	Controls the verbosity level. If used in a configuration file, please, supply an Integer level. If used as a command line argument, you can pass -v or –verbose multiple times, so that the verbosity level will be the number of -v and –verbose read from command line. See valid levels and their meaning in **Table 3**. Note that this option is incompatible with quiet.	**0**

Table 2: Global PyAutoFEP options accepted by all scripts

Option or argument	**Accepted values**	**Comment**	**Default**
quiet	Boolean	Suppress all output, but error messages. Note that this option is incompatible with verbose.	**Off**

1.3.2 Output and troubleshooting

PyAutoFEP prints information about the execution to the console. How much is printed is controlled by verbose and quiet options. The following verbosity levels are available.

Table 3: Verbosity levels used in PyAutoFEP output

Level	**What will be printed**	**Notes**
-1	**[ERROR]**	Quiet level. Nothing, just critical errors (which are almost always fatal and will halt the execution)
0	*DEFAULT* **[ERROR]**	Default level. Formatted output, displaying general information about the run and critical errors.
1	**[WARNING]** *DEFAULT* **[ERROR]**	All the above, plus warnings showing possible non-fatal problems found in the input files or during the execution. This is the recommended verbose level.
2	**[INFO]** **[WARNING]** *DEFAULT* **[ERROR]**	All the above, plus informational messages useful to identify problems during the execution.
3	**[DEBUG]** **[INFO]** **[WARNING]** *DEFAULT* **[ERROR]**	Debug level. Useful mostly for development purposes, will print lots of debug messages with deep details of the execution. Rdkit messages will also be printed.
4	*OPENBABEL* **[DEBUG]** **[INFO]** **[WARNING]** *DEFAULT* **[ERROR]**	Openbabel errors level. All of the above, plus turns on error logging from openbabel, which can be a LOT of messages. Only useful to inspect openbabel errors.

Messages are preceded by the following tags: **[ERROR]** , **[WARNING]** , **[INFO]** , and **[DEBUG]** . Note that default output, rdkit, and openbabel messages are not preceded by tags.

When something does not work as expected or does not work at all, one may try the following steps to identify the problem.

Table 4: Suggested problem solving steps

Step	Comment
Verify your input files	Make sure the files exists, formats are correct, and check for typos or incorrect line endings (if you are mixing OS)
Rerun with a verbosity level **2** or **3** (see above)	The printed info can hint for problematic parsing of input files, misplaced options or other problems.
Check the partial outputs	Sometimes the partial output files can help diagnose the error.
Fill a bug report at [LINK]	Sorry, you may have found a bug. Please, report to the issue tracker at GitHub so we can fix it.

1.3.3 Progress file

In order to safely save data between the runs and during each run, a progress file (a Python3 pickle file) will be used. The option progress_file is used to supply a progress file to any script. By default, no progress file will be read, but a temporary, hidden progress file will be created and used during the execution of a script.

The progress file is a Python dictionary containing relevant info for the current script that can be later accessed to save time and to communicate between modules. In general, it is and a good idea to use a progress file. It can save your time by safely carrying data from the perturbation map generator to the hybrid topology builder to the analysis script. Also, some pre-computed data (eg: MCS, input molecules) will be stored, speeding up the next module or another run of the same module. Previously saved data is not lost, but rather incrementally saved inside the progress file.

In some situations, a progress file will contain data that is incompatible with a modified option. This will be the case when you switch mcs_type option. When you do so, you have to consistently use the same progress file for either mcs_type = **graph** or mcs_type = **3d**. This also will be the case if you replace a molecule with another one with the same name. In this case, you will have to avoid using your old progress file.

1.3.4 MCS selection

PyAutoFEP implements two MCS selection algorithms: a graph-based MCS and a 3D-guided MCS, both relying on rdkit library functions. You can select between them using mcs_type. Graph-based MCS (mcs_type = **graph**) works in most cases and can handle rings smartly. This MCS algorithm will not break rings and will no match equal atoms with different valences. This is the default, and we recommend using it unless a perturbation is inverting a stereocenter.

3D-guided MCS (mcs_type = **3d**) is specially designed to be used when asymmetric centers are being inverted along the perturbation so that graph-based MCS would fail. 3D-guided MCS works by generating matching regions of the input molecules and using core-constrained flexible alignment to obtain conformations with maximum overlap. This procedure is done iteratively, expanding the matching region at each iteration. The algorithm can handle complex molecules, as carbohydrates and macrocycles.

2 Perturbation map generation (*generate_perturbation_map.py*)

2.1 Algorithms and perturbation maps

PyAutoFEP uses graphs to represent the perturbation maps so that maps are represented as connected, directed graphs, in which each node is a molecule and each edge is a perturbation. No loops are allowed (they would represent perturbing a molecule to itself, which is not supported). Currently, *generate_perturbation_map.py* does generate parallel edges (note: prepare_dual_topology.py actually accepts parallel edges in the form A→B, B→A). Three map types are available in *generate_perturbation_map.py*: star, wheel, and optimal (Fig). Each map type is associated with an algorithm to generate correspondingly the graph. Note that MCS between the pairs will be calculated and stored in the progress file **subsubsection 1.3.3** to subsequent use.

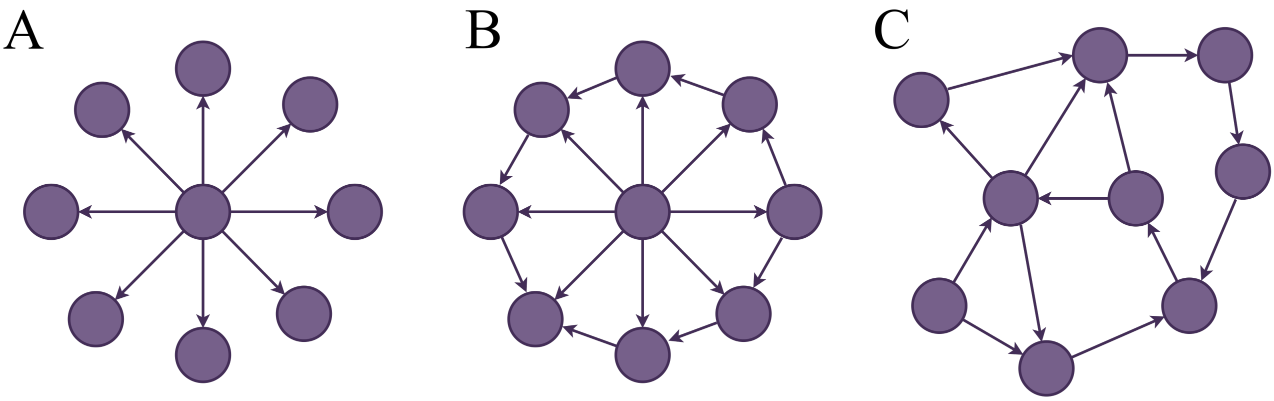


Figure 2: Three map types generated by *generate_perturbation_map.py*. (A) Star map, (B) Wheel map. (C) Optimal map.

A star map is a radial graph where all molecules are connected to a user-defined center molecule (2). You must provide one and only one center molecule using the option map_bias. The edges will be simply created connecting the center molecules to all other molecules.

A wheel map is similar to the star map, but non-center molecules are connected to two others, so all molecules belong to a closed cycle, and hysteresis analysis can be performed (2). You must provide one and only one center molecule using the option map_bias. *generate_perturbation_map.py* will, then, generate a star map exactly as described above. Then, the MCS between all non-center edges will be calculated. The number of perturbed atoms in a hypothetical perturbation between the edges will be calculated and used in the scoring function (See **subsection 2.2**). Finally, an And Colony Optimization algorithm will be used to find the best circular n-cycle graph connecting all outer nodes. Briefly, at each run a worker instance (ant) traverses the graph, choosing the edges to be used based on a probability given by the cost of the edge and the amount of deposited pheromone. When all nodes were visited, the ant finishes its run and deposits an amount of pheromone on all used edges. Probability pk(r,s) of choosing the edge r→s when the ant is in node r is given by (**ref: ADDREF DOI 10.1109/4235.585892**):

$$p_k(r,s) = \begin{cases} \dfrac{\tau(r,s)^\alpha \cdot \eta(r,s)^\beta}{\sum\limits_{u \in J_k(r)} \tau(r,u)^\alpha \cdot \eta(r,u)^\beta} & \text{if } s \in J_k(r) \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

Where $\tau(r,s)$ is the bias towards choosing $r \to s$ derived from the deposited pheromone, $\eta(r,s)$ is the bias towards choosing $r \to s$ derived from the cost of the perturbation, α and β are parameters to balance cost and pheromone effect on the probability and, $Jk(r)$ is the group of allowed edges from r.

The amount of pheromone effectively deposited by an ant run, $\Delta\tau_{k,\text{eff}}$, is given by:

$$
\begin{aligned}
\Delta\tilde{\tau}_k &= \frac{\left(\frac{\sum_{solutions} Tot_{cost}(j)}{N_{solutions}-1}\right)^{d_e}}{Tot_{cost}(solution_i)^{d_e}} \\
\Delta\tau_{k,\text{eff}} &= \begin{cases} max_d, & \text{if } \Delta\tilde{\tau}_k > max_d \\ \Delta\tilde{\tau}_k, & \text{otherwise} \end{cases}
\end{aligned} \tag{2}
$$

Where $\Delta\tau_k$ is the normalized amount of pheromone deposited, d_e is exponent for deposited pheromone (pheromone_exponent), $Totcost(j)$ is the total cost of the j-th solution, max_d is maximum amount of deposited pheromone (map_max_pheromone_deposited).

Default parameters for the scoring function will yield a (possible global) best solution minimizing the number of perturbed atoms.

Optimal maps are connected graphs optimizing the number of perturbations, the number of perturbed atoms, and the distance between any pair in the graph (**Figure 2**). *generate_perturbation_map.py* uses a modified ACO algorithm to obtain a graph minimizing cost function. This modified ACO starts from a complete graph and, at each step, an edge $r \to s$ is selected to be removed. The probability of choosing the edge $r \to s$ is given by:

$$
p_k(r,s) = \begin{cases} \frac{\tau(r,s)^{\alpha}\cdot\eta(r,s)^{\beta}}{\sum\limits_{(r\to s)\in\Omega*} \tau(r,u)^{\alpha}\cdot\eta(r,u)^{\beta}}, & \text{if } (r \to s) \in \Omega* \\ 0, & \text{otherwise} \end{cases} \tag{3}
$$

Where $\tau(r,s)$ is the bias towards choosing $(r \to s)$ derived from the deposited pheromone, $\eta(r,s)$ is the bias towards choosing $(r \to s)$ derived from the cost of the perturbation, α and β are parameters to balance cost and pheromone effect on probability, and $\Omega*$ is the group of allowed edges in the graph.

Each step, when an edge is removed, $\Omega*$ is updated with the available edges. If the removal of an edge would yield a graph with less than optimal_min_edges_per_node edges in any node, that edge is not removed. If optimal_min_edges_per_node is greater than or equal to **1**, a connected graph will be generated. If optimal_min_edges_per_node is greater than or equal to **2**, a connected graph with cycle closures will be generated.

The deposited pheromone equation (**Equation 2**) is equal to the one used on wheel maps. User can freely define all parameters of the scoring function **subsection 2.2**, and graphs with or without closed cycles can be generated.

ACO implementation is parallelized implementing the algorithm described in (`http://doi.dx/10.1109/MHS.2002.1058041`). However an arbitrary number of ants can be used to update the pheromone matrix (setting map_elitism). The number of parallel colonies (n_threads) and the number of runs before each communication step (map_communication_frequency) can also be freely set. Be careful about the balance between these values and the number of optimization cycles (map_runs). All ants will count towards map_runs, so using too few runs with a large number of parallel colonies (n_threads) and a large communication step (map_communication_frequency), means the pheromone matrix will be updated too few times and the cooperative effect will not be relevant. The automatic value for map_runs is, for instance, 20∗n_threads∗map_communication_frequency.

generate_perturbation_map.py can complete the few thousand optimization cycles (map_runs) required to generate a map with 20-30 nodes in under a minute on desktop hardware. Alternatively, the user can input custom maps to *prepare_dual_topology.py* without using the automated map generation at all.

2.2 Scoring function

Total cost of a solution for the perturbation map, Tot_{cost} in **Equation 4**, is given by:

$$
\begin{aligned}
L_{cost} &= \sum_{node_i} \sum_{node_j} (d_{ij})^{l_e} \\
D_{cost} &= e_m \sum_{node_i} (f_{deg}(node_i) - c_d)^{e_p} \\
P_{cost} &= p_m \sum_{edge_i} (f_{cost}(edge_i))^{p_e} \\
Tot_{cost} &= L_{cost} + D_{cost} + P_{cost}
\end{aligned}
\tag{4}
$$

Where L_{cost} is the length cost, d_{ij} is the distance between nodes i and j, l_e is length cost exponent (optimal_length_exponent), D_{cost} is the degree cost, $f_{deg}(node_i)$ is the degree of node i (ie: the number of edges connected to node i), c_d is the degree target constant (optimal_degree_target), e_m is the degree cost multiplier (optimal_degree_multiplier), e_p is the degree cost exponent (optimal_degree_exponent), P_{cost} is the perturbation cost, p_m is the multiplier (optimal_perturbation_multiplier), $f_{cost}(edge_i)$ is the cost of edge i (default: the number of atoms perturbed during the correspondingly perturbation), and p_e is perturbation exponent (optimal_perturbation_exponent). On classic ACO runs to generate wheel maps, L_{cost} and D_{cost} are set to **0.0**, and p_m and p_e are set to **1.0**. On modified ACO runs to generate optimal maps, all parameters can be freely chosen.

2.3 Script options

generate_perturbation_map.py accepts the following options or arguments, besides the globals (See **Table 2**).

Table 5: Options used by *generate_perturbation_map.py*

Option or argument	**Accepted values**	**Comment**	**Default**
input	Valid input files	Molecules to be used to generate perturbation map	
use_hs	Boolean	Take hydrogens in account when scoring perturbations.	**Off**
custom_mcs	SMARTS string or dictionary	Use this/these custom core instead of calculating MCS between pairs. If a single SMARTS is provided, it will be used for all pairs. If a dictionary is provided, only pairs present in the dictionary will use the supplied custom core and for all other pairs, the MCS will be calculated. The dictionary, if used, must be in format **{("mol_name_a", "mol_name_b"): SMARTS}** or **"mol_name_a-mol_name_b": SMARTS** (or other equivalent formats resulting in the same data structure. See **subsection 1.3**).	
map_type	star, wheel or optimal	Select type of map to be generated. See **subsection 2.1**	**optimal**
map_runs	Integer	Relevant for wheel and optimal maps. Run this many ants in ACO or modified ACO algorithm. Use **-1** to select the default value, 20∗n_threads∗map_communication_frequency. See **subsection 2.1**	**-1**

Table 5: Options used by *generate_perturbation_map.py*

Option or argument	**Accepted values**	**Comment**	**Default**
map_communication_frequency	Integer	Relevant for wheel and optimal maps. Run this many rounds of parallel Ant Colonies before selecting the best ants and synchronizing the pheromone matrix. See **subsection 2.1**	**20**
map_bias	Molecule name	For star and wheel maps, this molecule will be the center molecule. For these maps, you must supply one, and only one, molecule to be the center. For optimal maps, the map will be biased toward this molecule or these molecules. In this case, no molecule, or one or more molecules can be supplied.	
map_alpha	Float	Relevant for wheel and optimal maps. Pheromone biasing exponent. Controls the effect of the pheromone on the desirability of an edge. See **Equation 2**	**1.5**
map_beta	Float	Relevant for wheel and optimal maps. Perturbation cost biasing exponent. Controls the effect of the perturbation cost on the desirability of an edge. See **Equation 2**	**1.0**
map_pheromone_intensity	Float, x ≥ 0	Relevant for wheel and optimal maps. The intensity of deposited pheromone. Setting to 0 will disable pheromone, effectively defeating the purpose of the algorithm. See **Equation 2**	**0.1**
map_evaporating_rate	Float	Relevant for wheel and optimal maps. How fast the pheromone evaporates. When communicating, pheromone on all edges will be evaporated based on map_evaporating_rate See **subsection 2.1**	**0.3**
map_min_desirability	Float, x ≥ 0	Relevant for wheel and optimal maps. Minimal desirability of an edge. Larger values may lead to slower convergence, but more sampling.	**0.1**
map_max_pheromone_deposited	Float, x >0 or x = -1	Relevant for wheel and optimal maps. Caps the amount of deposited pheromone to this value. See **subsection 2.1**	**-1**
map_elitism	Integer, $x \geq$ **1** or $x =$ **-1** - or - Float, **0.0** $\leq x \leq$ **1.0**	Relevant for wheel and optimal maps. Use this many best solutions from each parallel colony to update pheromone matrix. If is a Float and 0 <x <1, it will be taken as a ratio and x*map_communication_frequency best solutions will be used If x >= 1, x best solutions will be used. If x = -1, all map_communication_frequency solutions will be used. See [2.1]	**-1**
optimal_max_path	Integer, x ≥ 1	Relevant for optimal maps. Distance between any two nodes on the graph must be at most this value.	**Off**
optimal_perturbation_multiplier	Float	Relevant for optimal maps. Multiplier for perturbation score. Setting this to 0 will nullify perturbation cost contribution to total score. See [2.2]	**1.0**

Table 5: Options used by *generate_perturbation_map.py*

Option or argument	**Accepted values**	**Comment**	**Default**
optimal_perturbation_exponent	Float	Relevant for optimal maps. Exponent for perturbation score. See [2.2]	**Off**
optimal_degree_target	Integer	Degree target. If set, map will be biased towards this much edges per node. Setting this to None (the default) will use optimal_min_edges_per_node as target. See [2.2]	**None**
optimal_degree_target	Float	Multiplier for degree cost. Set this to 0 to disable degree cost. See [2.2]	**0**
optimal_degree_exponent	Float	Exponent for degree cost. See [2.2]	**1.5**
optimal_min_edges_per_node	Integer	Each molecule must be part of at least this many perturbations. No solution violating this will be generated. Setting this to a value lower than 1 may lead to a disconnected map. Setting this value greater than 1 will generate a map with cycle closures. See [2.1]	**1**
optimal_min_edge_beta	Float	Extra edge beta parameter. Larger values increase the likelihood of more edges than the minimum amount.	**1**
optimal_unbound_runs	Integer, x >0 or x = -1	Relevant for optimal maps only. The number of runs when all edges can be removed. -1: edges can be removed on all runs. When an edge reaches a threshold (optimal_permanent_edge_threshold), it will not be selected by the ant either to be removed, becoming static. It will speed up the algorithm, at expense of an increased probability of reduced sampling.	**-1**
optimal_permanent_edge_threshold	Float, x >0	Relevant for optimal maps only. Edges with this much pheromone become static. -1: edges never become static. Setting this will speed up the execution but may reduce sampling.	**-1**

3 Dual topology generation and system building (*prepare_dual_perturbation.py*)

3.1 General workflow

Figure 3 describes the workflow of *prepare_dual_topology.py*. First, the script reads the input molecules and starting poses. The input molecules will be aligned to the poses so that atom ordering, missing hydrogens, and alternative coordinates will be corrected. Then, for each perturbation, a hybrid topology will be prepared, by reading the topology of both molecules and combining them into a new one. If you selected automatic system building, Gromacs tools will be used to prepare a topology for the receptor, and solvate and add ions to the complex. If you supplied a pre-solvated system, the ligands will be inserted in the binding site according to the starting poses read. In both cases, water legs will be automatically generated. Finally, the directories and files required to equilibrate and run the perturbations will be created, along with scripts to automate the run.

Figure 3: Detailed workflow of *prepare_dual_perturbation.py*

3.2 Starting poses readers

prepare_dual_topology.py requires ligand starting poses (from poses_input) to generate the ligand-receptor complexes and prepare a system to be run. The ligand poses can be read from MOL, MOL2, and PDB formats. (Note that, because PDB format is limited in the description of bonding and valence, assumptions will be made. Users are advised to check the resulting poses.). The pose can, then, be aligned to a reference receptor poses_reference_structure via the receptor's Cα. If the protein sequence of the reference structure and the pose structure differs, the sequences will be aligned prior to 3D alignment. The following readers are available.

3.2.1 PDB reader

PDB (pose_loader = **pdb**) reader will read one PDB file for each ligand. This file must contain a receptor and the ligand (and possibly other species). Openbabel is used to process the file. This reader is tolerant to some problems in the PDB file, (eg: missing side chains, non-standard atom naming, missing connect info), but alignment will fail for files with multiple occupancies. The starting pose for each ligand will be the PDB file pose, after alignment to the reference structure. Therefore, for this reader, poses_reference_structure is not used.

3.2.2 Autodock4 reader

Autodock4 reader (pose_loader = **autodock4**) can parse autodock4 output files and select clusters from it. It will automatically take care of atom renaming and reordering and can read ligands with or without merged hydrogens. This loader requires the receptor PDBQT file to align to the reference structure. This loader will try to read one autodock4 result per ligand and the starting pose for each ligand will be the docking pose, after alignment to the reference structure (poses_reference_structure).

3.2.3 Generic reader

The generic pose reader (pose_loader = **generic**) reader will read one small molecule file for each ligand. mol2 and mol files are read through RDKit and other formats are transparently read and converted using Openbabel. This file must contain only the ligand - no receptor will be read from it. The starting pose for each ligand will be the read pose.

3.2.4 Superimpose reader

The core-constrained ligand superimposer (pose_loader = **superimpose**) is a simple routine to align ligands to a reference. The superimposer will read a pose for a reference ligand (poses_reference_pose_superimpose) and all ligands will be superimposed to the reference using flexible alignment to the MCS. You can use mcs_custom_mcs to force a custom core instead of automatic MCS calculation. Note that poses_input is not used, as input molecules are read from input_ligands and/or from the progress file (See **subsubsection 1.3.3**).

3.3 Topology and force-fields

After preparing the poses for all ligands, *prepare_dual_topology.py* will read topologies for these molecules. PyAutoFEP does not parametrize molecules but is able to read output from common tools that do so, as AcPYPE, CGenFF, and LigParGen. Receptor topologies can be read or generated using **gmx pdb2gmx** (See **subsection 3.4** and **subsection 3.5**).

Both structures and topologies can be supplied using input_ligands. A structure file is a mol, mol2, or PDB file describing the 3D coordinates of the molecule and the connectivity. A topology file is a correspondingly Gromacs-compatible top, it, or atp (or any extension, actually) file. Data can be passed to input_ligands in the following ways. Missing files will be handled smartly (see below). Both structures and topologies can be supplied using input_ligands. A structure file is a mol, mol2, or PDB file describing the 3D coordinates of the molecule and the connectivity. A topology file is a correspondingly Gromacs-compatible top, itp, or atp (or any extension, actually) file. Data can be passed to input_ligands in the following ways. Missing files will be handled smartly (see below).

- *A file containing ligands names, structure, and topology files.* One name, followed by structure and topology, per line. Structure and topology will be guessed from the extension.
- *A list of lists containing ligands names, structure, and topology files.* One name, followed by structure and topology, per subitem. Structure and topology will be guessed from the extension.
- *A dictionary containing ligands names, structure, and topology files.* The dictionary key must be a ligand name and the value must be a list of structure and topology files. The order of structure and topology will be guessed. Eg: **"Benzene: benzene.mol2 benz_atoms.atp benz.top"**.See **subsection 1.3** for formatting guide.
- *Path to a directory containing files.* Files named *ligand_name.ext* will be read. All files with the same name (ignoring the extension) will be added to a ligand with that name (Eg: *benz.top*, *bezn.mol2*, and *benz.atp* would be added to a ligand named benz). Structure and topology will be guessed from the extension.
- *Path to a directory containing directories.* A ligand will be created for each directory inside input_ligands and all files inside it will be added to that ligand. The name of the ligand will be the name of the subdirectory. Structure and topology will be guessed from the extension. Eg: *benz/file.mol2* and *benz/top.top* would be added to a benz ligand.
- *A pickle file.* A dictionary (as described above) will be read from it.

Molecules will also be read from the progress file, if present. If data for a molecule is found in both the progress file and in input_ligands, the latter will be used. Missing fields from input_ligands will be read from the progress file, if present. So you can, for instance, supply the molecule structures using the progress file (this is the default if you are using a progress file created by *generate_perturbation_map.py*) and pass only the topologies using input_ligands.

After the *prepare_dual_topologies.py* reads topologies and structures, it will merge the topologies for each pair in the perturbation map (perturbation_map or read from progress file, see also **subsection 2.1**). The topology file loader is rather flexible and should be able to read and merge most Gromacs-compatible topologies. The MCS (See **subsubsection 1.3.4**) will be used to determine the common-core between start and end states. Then, a hybrid topology will be generated, representing the perturbation. The topology reader does have some limitations.

- *Online parameters on included files.* Currently, *prepare_dual_topologies.py* will not read **#include** from files, so if you have parameter files to your topology, you will have to pass it manually using input_ligands or the progress file.
- *N-body virtual sites.* Currently, N-body virtual sites are not supported. We expect to add it in a future version.
- *Virtual site positioning.* During manipulations of the structures, several force calculations and minimizations are performed. The small molecule force field employed, United Force Field, lacks a virtual site term. So a simple position restraint to the closest atom of the virtual site will be added to the molecule's force field. This will ensure that the optimized geometry at least makes sense. The actual positioning of the virtual site will not be precise, however. Note that this will likely be corrected during minimization in Gromacs, but the user is advised to make sure their structures are correct.

Along with the perturbation, atom charges and Van der Waals parameters will be scaled according to lambda_input. PyAutoFEP implements the following equation to construct the Hamiltonian of the perturbed state.

$$\mathcal{H}_{\lambda} = \lambda_{VdW,A}\mathcal{H}_{VdW,A} + \lambda_{VdW,B}\mathcal{H}_{VdW,B} + \lambda_{Coul,A}\mathcal{H}_{Coul,A} + \lambda_{Coul,B}\mathcal{H}_{Coul,B} + \mathcal{H}_{const} \tag{5}$$

Where $\mathcal{H}_\lambda$ is the total Hamiltonian of the λ state, $\mathcal{H}_{term,s}$ is the Hamiltonian from the contribution of the term *term* over atoms of state s (either A or B), $\lambda_{term,s}$ is scaling factor for the term *term* over atoms of state s, and $\mathcal{H}_{const}$ is the Hamiltonian of the unperturbed region.

From **Equation 5** is clear that the contribution of Van der Waals and Coulomb terms for states A and B are not coupled and can be varied arbitrarily. Note that although λ should be a number such as $\lambda \in [0.0, 1.0]$, even this restriction can be lifted by running with no_checks. To allow the equation **Equation 5** to be used, PyAutoFEP does not use Gromacs FE code, but rather directly generate several tpr run files for each state (See **Figure 1**). Also, from **Equation 5**, bonded terms and masses are not currently perturbed.

3.4 Automated system builder

The automated system builder is a very simple function which uses **gmx pdb2gmx**, **gmx editconf**, **gmx solvate**, and **gmx genion** under the hood. It automates the building of a simple receptor-ligand system and will fail for complex cases (eg: membrane systems, systems with vacuum regions, solvent mixtures, and possibly systems with nucleic acids on cofactors). In such cases, please, use the system loader (**subsection 3.5**). Automated system building is the default running mode and will be used unless pre_solvated is used. Options relevant to the system builder are grouped under the prefix buildsys_*. The following steps will be executed by the system builder.

- **gmx pdb2gmx**. Run to prepare the receptor topology and a corresponding structure file. Force fields from Gromacs' default library directory can be selected using buildsys_forcefield. Custom force fields can be used by providing a force field directory to extradirs and using buildsys_forcefield**=1** (the default).
- Add small molecule to the system, according to the staring pose read (**subsection 3.2**). Both the structure and topology will be edited. Ligand parameters will be added to the system topology and position restraints to the protein and to the ligand will be generated.
- **gmx editconf**. A simulation box will be generated. Size can be set via buildsys_watershell options.
- **gmx solvate**. The system will be solvated to fill the box size from the previous item. PyAutoFEP will try to guess the water model. buildsys_water option can be used to pass the **-cs** option to **gmx solvate**, but be aware that non-water solvation is not currently supported.
- **gmx genion**. Counterions will be added to the system. The default is to neutralize the system, but an ionic concentration can be passed with buildsys_ionconcentration. Cationic and anionic species can be given with buildsys_pname and buildsys_nname, respectively.
- A index file will be generated. Whether it will be done using **gmx genion** depends on selection_method. You can supply custom index groups with index_string.

3.5 System loader

As an alternative to system building, a fully prepared system, and topology can be read. Currently, this is required for simulating systems containing a membrane and/or cofactors and using an equilibrated system as the starting point to the FEP.

The system structure will be read from a PDB file with the option structure. Header (including CRYST1) and atom positions (ATOM and HETATM) will be read. Note that a CRYST1 line is required by GROMACS to set up the simulation box. Only a single model can be read from the file. The protein/receptor has to be the first entry.

The topology file will be read (option topology) and edited to add the ligand and to reflect (possible) changes in the number of waters and ions. **#include** directives will not be followed, so your [system] must be described in this file. You can omit the topology file by running with no_checks, but this will require you to manually edit/generate the topology afterward.

A GROMACS-compatible index file (.ndx) can be optionally read (index) so that the protein atoms will be read from it. A group called Protein (case-sensitive) is expected on the index file. If such file is not provided, **gmx makendx** will be used to guess the groups present in the structure file.

After all the files are read, the ligand will be added to the system immediately after the protein, according to the read pose (**subsection 3.2**). Water molecules clashing with the ligand will be removed. The option pre_solvated_radius controls the cutoff for this removal. The topology file topology will be updated to reflect the changes.

3.6 Solvent leg system builder

Regardless of using the Automated system builder (**subsection 3.4**) or the System loader (**subsection 3.5**), a system containing the ligand in solution will be built to run the solvent leg of the FEP. The following steps will be run.

- A PDB containing only the ligand will be prepared.
- **gmx editconf**. A simulation box will be generated using the size given via buildsys_watershell.
- **gmx solvate**. The system will be solvated to fill the box size from the previous item. PyAutoFEP will try to guess the water model. buildsys_water option can be used to pass the **-cs** option to **gmx solvate**, but be aware that non-water solvation is not currently supported.
- **gmx genion**. Counterions will be added to the system to neutralize the system only. Cationic and anionic species can be given with buildsys_pname and buildsys_nname, respectively.
- A index file will automatically be generated.

3.7 Output options

The final output of *prepare_dual_topologies.py* will be ready-to-run GROMACS input files along with directories and scripts to simplify the submission. The following general directory structure will be used:

[**base_directory**] Root directory for this FEP series, set using **perturbations_dir**

- [**ligand1-ligand2**] Data for the perturbation ligand1→ligand2
  - [**protein**] Data for the complex leg of ligand1→ligand2 FEP.
    - [**build_system_*hhmmss_ddmmyyyy***] Directory containing files used during the building of the system used to run the complex leg
      - **...** Log files, intermediate topologies, and PDB, GROMACS outputs
    - [**lambda0**] Minimization and equilibration files for $\lambda = 0$ window. Actual directories will vary depending on the minimization and equilibration scheme for complex leg (complex_mdp or template_mdp). Furthermore, files supplied in extrafiles and directories given in extradirs will be copied to this dir.
      - [**build_system_*hhmmss_ddmmyyyy***] Symlink to this directory on parent folder
      - [**md**] MDP files that will be used to generate TPR files for the FEP production run. After the minimization and equilibration, a copy of these very TPR files will be placed here.
      - [**min01**] Data for first minimization step, name may vary
      - [**min02**] Data for second minimization step, name may vary
      - [**nve**] Data for NVE equilibration step, name may vary
      - [**nvt**] Data for NVT equilibration step, name may vary
      - [**npt**] Data for NPT equilibration step, name may vary
      - **FullSystem.pdb** Symlink to the PDB structure file in the parent directory, name may vary as given by structure.

- **SystemFull.top** This may be either a symlink to the topology in the parent directory or a complete topology file. The former option is the default when using REST or REST2. The name will be given by topology
- **index.ndx** Symlink to index file in the parent directory, name given by index
- **ligand.atp** Ligand atom type data for $\lambda = 0$
- **ligand.itp** Ligand topology for $\lambda = 0$
- **posre_*.itp** Symlinks to position restraint files in the parent directory

[**lambda1**] Minimization and equilibration files for $\lambda = 1$ window

[**...**]

[**md**] Directory containing the data for the production run of the ligand1→ligand2, complex leg

- [**analysis**] Data generated during the analysis step
  - [**lambda0**] Output files fo the analysis of the $\lambda = 0$ production MD. XVG files for the ligand and protein RMSD, RMSF, SASA, and center of mass distance.
  - [**lambda1**] Analysis data for $\lambda = 1$
  - **...**
- [**lambda0**] Data generated during the production run for $\lambda = 0$
- [**lambda1**] Data generated during the production run for $\lambda = 1$
- **...**
- [**rerun**] Data generated during the rerun step
  - **lambdaX.xvg** Pressure and volume per frame from original MD
  - **rerun_structX_coordY.edr** GROMACS energy file from the rerun using coordinates from $\lambda = X$ window and Hamiltonian from $\lambda = Y$ window
  - **rerun_structX_coordY.xvg** Potential energy per frame from the rerun using coordinates from $\lambda = X$ window and Hamiltonian from $\lambda = Y$ window
  - **unk_matrix.pkl** Reduced potential matrix extracted from the EDR files.

FullSystem.pdb PDB file for the starting positions for the complex leg system, name may vary as given by structure

index.ndx GROMACS-compatible index file, will containing autogenerated groups and groups from index_string, name given by index

SystemFull.top GROMACS-compatible topology file for the full system, name given by topology. This file will only be used if the topology file inside each lambda dir is a symlink, ie, when no REST or REST2 is used.

posre_*.itp Position restraint files for each macromolecule segment in the system

[**water**] Data for the solvent leg of ligand1→ligand2 FEP.

- [**...**] (Same directories as the complex leg)

runall_*ligand1-ligand2*_protein.sh Script to run the complex leg of the perturbation.

runall_*ligand1-ligand2*_water.sh Script to run the solvent leg of the perturbation.

[***ligand2-ligand3***] Data for the perturbation ligand2→ligand3. The name will vary.

[***ligand3-ligand4***] Data for the perturbation ligand3→ligand4. Name will vary.

[**...**]

pack.sh A script used to compress data after the execution all FEP calculations.

runall.sh A script containing all the or submission commands for all perturbations.

progress_data.pkl Progress file for the project, used during analysis.

By default, *prepare_dual_topologies.py* will generate a self-extracting bash script using tar. Alternatively, output_packing can be passed to select a uncompressed directory (output_packing=**dir**) or a regular tgz file (output_packing=**tgz**). After all calculations are done, *pack.py* will be run to create a compressed tgz file, ***base_directory*.tgz** (name given by perturbations_dir). This file will contain [**analysis**] and [**rerun**] directories, the log and a xtc file, for all perturbations. Note that, by default, this xtc file will be PBC-corrected and centered using **gmx trjconv**, as well as subsampled in frames.

By default (output_scripttype=**bash**), *runall.sh* will be a regular bash script to sequentially minimize, equilibrate, run and rerun MD, then run analysis (**Figure 1**). Alternatively, *runall.sh* can be a submission script for PBS/Torque (output_scripttype=**pbs**) or Slurm (output_scripttype=**slurm**), in which case each complex and water leg will be submitted as a single job to the scheduler, and a last, packing job will be submitted depending on the completion of all legs. Using **output_submit_args** will add command line arguments to the submission command. **output_resources** can be used to fine tune the submission of run jobs (using **all** prefix) and the pack job (using **pack** prefix). **output_resources** will read a dictionary in the following form: "***prefix _resource*:value;*prefix_resource*:value**". Currently, four resources will be recognized:

Table 6: Options for fine tuning the resource requiremnts of jobs submitted by the run script

Resource	Description	PBS/SLURM options	Default value for perturbation jobs	Default value for pack jobs
nodes	How many nodes to be used for each job	#SBATCH –nodes #PBS -l nodes	**1**	**1**
cpu	How many CPU to be used in each node	#SBATCH –ntasks-per-node #PBS -l ppn	**12**	**1**
gpu	How many GPU to be used in each node	#SBATCH –gres:gpu #PBS -l gpus	**1**	**0**
time	Maximum job runtime, in hours. Make sure to use a integer value, including - or : may lead to unexpected results	#SBATCH –time #PBS -l walltime	**24**	**1**

Options output_runbefore and output_runafter can be used to provide commands to be executed at the beginning of the run script and at its end, so before and after the actual GROMACS and analysis commands. This can be useful to load required modules on the run node (output_runbefore) and to perform cleaning/logging actions (output_runafter). output_runbefore can also be used to modify the PATH so that commands (eg, parallel) can be found.

Finally, output_scripttype accepts any string value, so that will be used as a submission (or run) command. Using a custom command requires output_template pointing to a file that will be used as the header for the submission script. No substitutions will be performed on this file. Using output_scripttype and output_template is useful to prepare submission scripts to other schedulers than PBS or SLURM.

Regardless of which output_scripttype is used, the same steps will be executed by the run script *runall_ligandX-ligandY_protein.sh* or *runall_ligandX-ligandY_water.sh*. The single exception is analysis, which will only be run for the complex leg.

Table 7: Steps executed during EFP run in run node

Step	Description	Parallel execution
Minimization	The specific minimization jobs will vary, depending on complex_mdp and water_mdp. By default, three minimization steps will be run on the complex leg (min01, min02, and min03) and two on the water leg (min01 and min02). The first one, run only in the complex leg, will minimize the solvent while restraining the ligand and the protein. The second and third will minimize the while system using emtol=1000.0, then emtol=100.0 kJ/mol*nm.	No, each λ window will be minimized on its own.

Table 7: Steps executed during EFP run in run node

Step	**Description**	**Parallel execution**
Equilibration	The specific equilibration jobs will vary, depending on complex_mdp and water_mdp. By default, three equilibration steps will be run both on the complex leg and on the water leg (NVE, NVT, and NPT). First, a 200 ps NVE with protein and ligand heavy atoms restrained, or no restraints for the water leg. Then, a unrestrained 200 ps NVT equilibration using a velocity rescaling thermostat will be run. Finally, a 200 ps NPT equilibration using a Parrinello-Rahman isotropic pressure coupling, and no restraints will be used.	No, each λ window will be equilibrated on its own.
Run	All λ windows will be executed in parallel using GROMACS **mpirun gmx mdrun -multidir**. All available CPU and GPU, as detected by gmx, will be used and one MPI process per λ window will be used. In case of continuations, if HREX is used, all the λ windows must be synchronized on the same frame, otherwise, GROMACS will die with errors.	Yes, complete parallelization using GROMACS code **-multidir**
Rerun	Each trajectory obtained using different λ windows will be reevaluated using all Hamiltonians for all λ windows. The reevaluation will be done using **gmx mdrun -rerun** and a .edr file will saved to rerun directory. The rerun step will run **gmx** N^2 times, where N is the number of trajectories (ie, the number of λ windows). After all .edr files are ready,	Yes, using GNU parallel. The number of parallel jobs can be supplied using output_njobs. By default, the run script will detect the system layout and run a job per GPU or a job per 2 CPU, if no GPU is present.
Analysis *(Only on complex legs)*	First, **gmx trjconv** will be used to generate a PBC-corrected, centered trajectory, skipping 100 frames· The resulting trajectory will be saved in GROMACS compressed format (XTC). Then, **gmx rms**, **gmx rmsf**, **gmx distance** and **gmx sasa** will be used to calculate RMSD, RMSF, protein-ligand distance and ligand SASA. Results will be saved to **analysis/lambdaX** directory.	Yes, using GNU parallel. A job per λ will be run.
Pack	A pack job will be run only once, after all jobs are completed. The pack job will use **tar** to create a compressed file containing [**analysis**] and [**rerun**] directories, the log and a xtc file, for all perturbations. The same directory structure described above will be used, and the file will contain .xvg, .edr, .tpr, .xtc, and .pkl files.	No. Currently, a **tar czf** command will be used.

During the **analysis** step, the potential energy will be read from the edr files from the **rerun** step. **gmx energy** will be used to generate the xvg files, which will be read to prepare the reduced potential matrix. This can be done using a Python script (output_collecttype=**python**) or a compiled binary (output_collecttype=**bin**, the default), or any command (output_collecttype=[**script_name**]). The default option is to use a binary created using PyInstaller and linked to a old version of the glibc, so it should run on most of the systems. In case the python script is selected, Python3 executable must be in $PATH during runtime (See output_runbefore), and only modules from the Python Standard Library will be used. output_collecttype also accepts any program or script name to be used as the collect executable. The command line will be **”*script_name* –temperature *absolute_temperature* –gmx_bin *path_to_gmx* –input **.xvg*”**, where ****.xvg*** will be all the xvg files from the **collect** step.

3.8 Script options

Option or argument	Accepted values	Comment	Default
topology	Valid GROMACS-compatible topology file	If pre_solvated=**True** (ie, a solvated system is provided), the system topology will be read from this file. If pre_solvated=**False** no topology will be read. In both cases, the value of topology will be used as the name of topology files created during the setup.	**FullSystem.top**
structure	Valid PDB file	If pre_solvated=**True**, the solvated system will be read from this file. If pre_solvated=**False**, the receptor structure will be read from this file. See **subsection 3.4** and **subsection 3.5** for further info.	*No default: required option*
index	Valid GROMACS-compatible index (NDX) file	If pre_solvated=**True**, this file will be read and used as an index file. The value of index will be used as the name of index files created during the setup.	**index.ndx**
index_string	Dictionary	A dictionary containing index groups in the format "***group1_name:selection_algebra***". These groups will be added to the index file (index). See also selection_method.	**None**
selection_method	mdanalysis, internal	Method to be used when handling atom selections and indexes. With selection_method=**mdanalysis**, the mdanalysis module will be used to process index_string and to select water molecules to be removed, if pre_solvated=**True**. Using mdanalysis allows for more a flexible selection algebra. With selection_method=**internal**, **gmx make_ndx** will be used to create the index file and a internal method will be used to select water molecules.	**internal**

Option or argument	Accepted values	Comment	Default
input_ligands	Path to a file; a path to a directory; dictionary	If a file is supplied, ligands names, mol2, and topology files will be read from the file lines. Ligand's name must be the first entry and subsequent columns will be guessed. If a directory is given, this directory must either contain files named ***ligand_name.mol2*** and ***ligand_name.top***, or directories named after the ligands containing mol2 and top files. If a dictionary is given, it must be in the format "ligand1_name:ligand1_name.pdb,ligand1_name.top;ligand2_name:ligand2_name.pdb,ligand2_name.top". When reading files, the file type will try to be guessed by extension: mol, mol2, and pdb files will be taken as ligand structures; top, itp, atp, and any other type will be read as topology. Note that data absent from this option will be read from the progress file. Ligand molecules and topologies read from this option will be saved, possibly updating data, to the progress file. *prepare_dual_topology.py* requires at least a structure (mol, mol2 or pdb) and a topology file (top, itp, etc) for each ligand in the perturbation map (perturbation_map). Multiple topology files per ligand can be supplied, but, currently, they will be concatenated to the same topology file in the same order as read. See **subsection 3.2** for info about how the ligands are processed.	**None** *(So that all data will be read from the progress file, otherwise this is a required option.)*
perturbation_map	Text file	Perturbation map to be used as the source for perturbations. By default, the perturbation map is read from the progress file, but giving this option will super seed the map from the progress file, if any. The map file must be a column file, with a perturbation per line. The first ligand name read (first column) is the state A ligand, and the second ligand (second column) is the state B ligand. An optional, third column, may be used to select a lambda scheme for this particular perturbation, superseeding lambda_input. Lines starting with ";" or "#" will be regarded as comments and ignored. Note that if no perturbation map is read from the progress file nor from this option, the script will fail.	**None** (By default, a perturbation map will be read from the progress file)
perturbation_dir	*String*	Base name for the pertubation directory structure (See **subsection 3.7**). This will also be the filename for the compressed file containing the pertubation directory structure, if output_packing is either **tgz** or **bin**.	**perturbations_*hhmmss*_*ddmmYYYY***
extrafiles	*list*	A list of extra files to be copied to the system building and perturbation directories. *prepare_dual_topology.py* will not process or modify these file in any fashion. By default, all required files will be generated on the fly. This option allows the user to use custom topologies, index files, etc.	**None**, no extra files will be copied.

Option or argument	Accepted values	Comment	Default
extradirs	*list*	A list of extra directories to be copied to the system building and perturbation directories. This option is akin to extrafiles, but for dirs. Using a force field not included in your GROMACS distribution can be easily done by passing the GROMACS-compatible force field directory (in general, *force_field_name.ff*) to this option.	**None**, no extra dirs will be copied.
gmx_maxwarn	*int*	This value will be directly passed to GROMACS *gmx grompp -maxwarn* during system setup and during the run steps (ie, it will be added to the run scripts). By default, a *-maxwarn*=**1** is used, because the intermediate windows may have non-integer total charges.	**1**
gmx_bin_local	*String*	Path to the local GROMACS binary. The local *gmx* will be used to during the system setup to run *pdb2gmx*, *editconf*, *grompp*, *makendx*, and *genion*. The path should be a valid binary path in the local machine (ie, the machine PyAutoFEP is running). By default, a *gmx* binary in the path will be used.	**gmx**
gmx_bin_run	*String*	Path to the GROMACS binary to be used to run the MD. The run *gmx* will be used during the minimization, equilibration, run, rerun, and analysis to run *grompp* and *mdrun*. The path should be a valid binary path in the run node. By default, a *gmx* binary in the path will be used.	**gmx**
mcs_custom_mcs	*String* or *list*, valid SMARTS	Custom SMARTS representing a Maximum Common Substructure to be used instead of MCS calculation. Supplying a *String*, ie, a single MCS will cause *prepare_dual_topology.py* to use that SMARTS as the common core for all perturbations and small molecules alignment, if pose_loader=**superimpose**. A list can be used to supply a custom core for specific molecule pairs. To do such, the format must be: "**mol1, mol2, smarts1, mol3, mol4, smarts2**", so that **mol1** and **mol2** will have **smarts1** as core.	**None**
mcs_custom_mcs	*String* or *list*, valid SMARTS	Custom SMARTS representing a Maximum Common Substructure to be used instead of MCS calculation. Supplying a *String*, ie, a single MCS will cause *prepare_dual_topology.py* to use that SMARTS as the common core for all perturbations and small molecules alignment, if pose_loader=**superimpose**. A list can be used to supply a custom core for specific molecule pairs. To do such, the format must be: "**mol1, mol2, smarts1, mol3, mol4, smarts2**", so that **mol1** and **mol2** will have **smarts1** as core.	**None**

Option or argument	Accepted values	Comment	Default
poses_input	*String* or *List* of mol2 or mol	Poses to be used as starting poses for the ligands. This can be either a file containing the file names, one per line, or a list or filenames. The name of the molecules read from the corresponding fields from molecule files, will be matched to the names in the perturbation map. If there is no molecule name in the file, the name will be read from the filename.	**None**. *Note: this is a required option for all pose loaders but* **superimpose**.
pose_loader	**generic**, **pdb**, **superimpose**, **autodock4**	Select poses format to be read. See **subsection 3.2**.	**None**
poses_reference_structure	*String*, valid macromolecular file.	Reference structure for poses_input. The poses read will be aligned to structure using the same translation/rotation used to align poses_reference_structure to structure. If the ligand poses are already aligned to the receptor in structure, this option is not needed. A complex system can be used here.	**None**. *By default,* *structure* *will be used. See also* *pose_loader* = ***pdb*** *above.*
poses_reference_pose_superimpose	*String*, valid PDB, mol, or mol2 ligand file.	The ligand pose to be used as source of constraints for pose_loader = **superimpose**.	**None**. *Required option for* *pose_loader* = ***superimpose***.
poses_advanced_options	*Dictionary*	Advanced options passed to the underlying functions used in the pose loaders. The following options will be passed (See rdkit documentation, Biopython documentation): **rdkit.AllChem.EmbedMolecule** maxAttempts=50, randomSeed=2342, useRandomCoords=True, clearConfs=True, ignoreSmoothingFailures=True, useExpTorsionAnglePrefs=True, enforceChirality=True, boxSizeMult=5.0 **rdkit.AllChem.EmbedMultipleConfs** maxAttempts=50, numConfs=50, randomSeed=2342, clearConfs=True, ignoreSmoothingFailures=True, useExpTorsionAnglePrefs=True, enforceChirality=True, boxSizeMult=5.0, numThreads=0 **rdkit.AllChem.ConstrainedEmbed** useRandomCoords=True, ignoreSmoothingFailures=True **Bio.pairwise2.align.globalds** seq_align_mat=blosum80, gap_penalt=-1	**None**

Option or argument	Accepted values	Comment	Default
buildsys _forcefield	*Integer*	Only applicable if pre_solvated=**False** (the default). Force field to be used during the system building. This value will be passed directly to *gmx pdb2gmx -ff*. The meaning of each number may vary depending on the GROMACS installation. A custom force field can be used by passing **1** (the default value) and supplying a force field directory to extradirs.	**1**.
buildsys _water	**default**, **spc**, **spce**, **tip3p**, **tip4p**, **tip5p**, **tips3p**	Only applicable if pre_solvated=**False** (the default). Water model to be used in the system building. If default, "**1**" will be supplied to *gmx pdb2gmx -water*, selecting FF's default water model, otherwise the value name will be passed as option to *-water*.	**1**.
buildsys _watershell	*Integer*	Distance, in Angstroms, between the solute and the box face in the complex leg if pre_solvated=**False**, and in the solvent leg. The value will be passed to *gmx editconf -d*.	**10 Å**.
buildsys _ionconcentration	*Float*	Only applicable if pre_solvated=**False** (the default). Concentration, in mol/L, of added ions to be added to the system on top of the ions needed to neutralize the system. Using **0.0** here will cause the adding only of the counterions. The value will be passed to *gmx genion -conc*.	**10 Å**.
buildsys _nname	*String*, valid values depend on the force field.	Name of the positive ion, passed to *gmx genion -nname*	**CL**.
buildsys _pname	*String*, valid values depend on the force field.	Name of the positive ion, passed to *gmx genion -pname*	**NA**.
pre_solvated	*Boolean*	Use a pre-solvated structure. See **subsection 3.4** and **subsection 3.5** for further info.	**False**.
presolvated _radius	*Float*	Remove water molecules this close, in Angstroms, to the ligand. Use **0 Å** here to avoid water molecule removal.	**1 Å**.
presolvated _protein _chains	*Float*	The number of distinct macromolecular chains in the pre-solvated system passed to structure. Ligand will be inserted immediately after the protein in the structure PDB, so this is required to adjust the topology accordingly.	**1**

Option or argument	Accepted values	Comment	Default
lambda_input	**lambda12**, **lambda24**, **lambda16wang**, **lambda23wang** or *Dictionary*	Selects the λ scheme to perturb molecule A to molecule B. Users can use one of the four schemes available (See **Table 11**, **Table 12**, **Table 13** and **Table 14**) or supply a dictionary (or file with the required structure) to use a custom one. If so, the dictionary must be formatted as follows: **vdwA: 1.00000, 1.00000, 1.00000; vdwB: 0.00000, 0.11850, 0.18978; coulA: 1.00000, 0.75000, 0.50000; coulB: 0.00000, 0.00000, 0.00000** Where **vdwA** and **vdwB** are the Van der Waals λ for states A and B, respectively, and **coulA** and **coulB** are the Coulomb λ for states A and B, respectively. The length of the lists for each of **vdwA**, **vdwB**, **coulA**, **coulB** must be equal, so that the λ value for the window n, λ_n, will be the $n+1$ element of the list. Note that PyAutoFEP does not impose constraints to the λ values, so any *Float* can be used.	**lambda12**
template_mdp	**default_nosc**, **charmm_nosc**	Use these group of mdp templates to setup minimization, equilibration, and run steps. The currently available templates for complex legs are only suitable to simple protein-ligand systems. The water leg templates, on the other hand, can likely be used for any ligand. The default mdp are described in **Table 7**. See also mdp_substitution, which can be useful in conjunction with this option.	**default_nosc**
complex_mdp	*List* of valid file names or a path to a file containing valid filenames	Use these files, in order, as mdp files for each step of the complex leg. Supersedes template_mdp for the complex leg only. The last file in the list must be named md.mdp and will be used as the mdp file for the run step (See **Table 7**).	**None**
water_mdp	*List* of valid file names or a path to a file containing valid filenames	Use these files, in order, as mdp files for each step of the water leg. Supersedes template_mdp for the water leg only. The last file in the list must be named md.mdp and will be used as the mdp file for the run step (See **Table 7**).	**None**
mdp_substitution	*Dict* or a path to a file containing substitutions	Replace or add the data in this *Dict* to the mdp files of both complex and water legs. There are two ways of using this option. All mdp files of both legs can be modified by using a *Dict* in the format: **mdp_parameter1: value1; mdp_parameter2: value2**. Keep in mind that the mdp for all steps will be modified. Alternatively, the mdps of specific steps can be modified using a dictionary in the format: **{”npt”: {”compressibility”: 3.72}; ”md”: {”compressibility”: 3.72}}**. This would change the compressibility only during the npt equilibration and run steps of both legs. See also md_temperature and md_length.	**None**
solute_scaling	*Float*	Apply solute scaling using this value as β	**None**

4 Result analysis (*analyze_results.py*)

analyze_results.py will read a compressed file created from *pack.sh* or the uncompressed data from this file, and use statistical estimators to obtain $\Delta\Delta G$ estimative for each perturbation. The perturbation map and the reduced potential matrices will be read from the progress file found in the data file or directory. *analyze_results.py* will fail if an unexpected directory structure is found. From the perturbation directories, water and complex perturbations will be read and BAR or MBAR estimators (**subsection 4.1**) can be used to calculate the $\Delta\Delta G_{solvent_{A\to B}}$ and $\Delta\Delta G_{complex_{A\to B}}$, and, from these, $\Delta\Delta G_{A\to B}$. Pairwise $\Delta\Delta G_{A\to B}$ can be transformed to $\Delta\Delta G_{A\to ref}$, where *ref* is a reference ligand in the perturbation map by summing the respective edges on the map (**subsection 4.2**). If Replica-Exchange is used, exchange probabilities and occupancy matrices can be read from GROMACS run log and graphs can be plotted (**subsection 4.3**).

4.1 Estimators

Currently, Bennett Acceptance Ratio (BAR) and Multistate-Bennett Acceptance Ratio (MBAR) (**ref:**) are the available estimators in PyAutoFEP. Both are used from pymbar implementation thorough alchemlyb (**ref:**). BAR uses data from two states to estimate ΔG using the equation 6.

$$\begin{aligned} \sum_{i=1}^{n_i} \frac{1}{1+\exp(\ln(n_i/n_j)+\beta\Delta U_{ij}\beta\Delta G)} \\ -\sum_{j=1}^{n_j} \frac{1}{1+\exp(\ln(n_j/n_i)-\beta\Delta U_{ji}+\beta\Delta G)} = 0 \end{aligned} \tag{6}$$

Where $\beta = \frac{1}{RT}$; n_i and n_j are the number of samples in each state; ΔU_{ij} is the difference between reduced potential in states i and j; ΔG is the free energy difference.

MBAR is an expansion of BAR using data from several states to estimate ΔG. It has been shown that MBAR yields the lowest variance between free energy methods (**ref**), and a estimator for the error is available. This error estimative will be printed together with the pairwise $\Delta\Delta G$. When MBAR is used (the default), a state-overlap matrix will be calculated and can be plotted to a graph (if analysis_types contains **overlap_matrix**).

MBAR uses equation 7.

$$\hat{G}_i = -\beta^{-1} \ln \sum_{j=1}^{K} \sum_{n=1}^{N_j} \frac{\exp\left[-\beta U_i\right]}{\sum_{k=1}^{K} N_k \exp\left[\beta\hat{G}_k - \beta U_k\right]} \tag{7}$$

Where $\beta = \frac{1}{RT}$; $\hat{G}_i$ and $\hat{G}_k$ are the free energies at states i and k; U_i and U_k are the reduced potentials at states i and k; N_k is the number of samples at state k.

4.2 Graph options

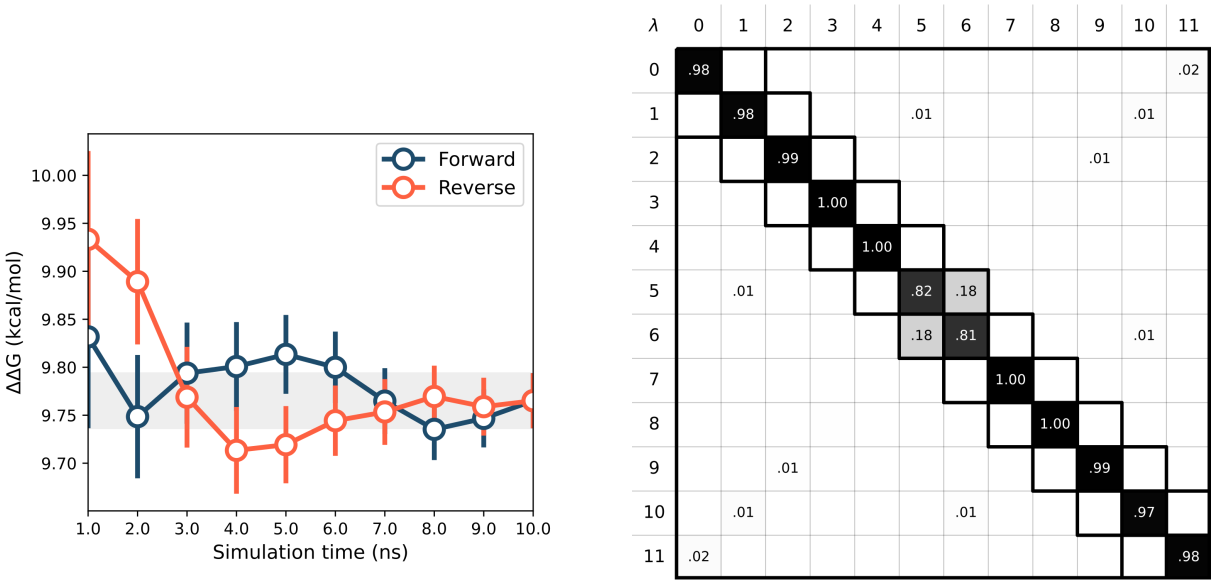
When analyzing the $\Delta\Delta G$ for a series, is often useful to use one of the molecules as the reference, so all $\Delta\Delta G$ are calculated with respect to it. PyAutoFEP implements four methods to sum $\Delta\Delta G$ along edges when more than one path is available. Whether different methods will yield different results, depends on the perturbation map used.

Using shortest path (center_ddg_method = **shortest**) will simply use the shortest path, in number of perturbations, to connect each molecule to the reference molecule. If multiple paths are available, only one will be used, as given by *networkx.single_source_shortest_path* (See relevant networkx documentation). This is the default mode and makes sense to be used when no alternative paths are present. Using shortest average path (center_ddg_method = **shortest_average**) will average all the shortest paths to the reference molecule and longer paths will be ignored. If a single path is available, it will be used.

Using all averages (center_ddg_method = **all_averages**) will average all possible paths connecting the molecule to the reference. A simple averaging will be used, so all paths will contribute equally to the $\Delta\Delta G$. Using weighted averages (center_ddg_method = **all_weighted_averages**), will work similarly to all averages, but the contributions of all paths will be weighted by the number of edges in a path.

4.3 Analysis types

Argument	**Comment**
replica_exchange	If during GROMACS MD run Hamiltonian replica-exchange was used using Plumed, PyAutoFEP can analyze the replica-exchange acceptance and replica occupancy. First, PyAutoFEP will read the replica-exchange attempts and accepted moves from GROMACS log file, reading for *Repl ex* lines. Then, the coordinates will be demuxed in respect to Hamiltonians. From the number of accepted moves, an empirical transition matrix will be constructed, normalized, and diagonalized. From the demuxed trajectory, the occupancy of each trajectory in each hamiltonian will be calculated. The occupancy and the Hamiltonians visited by each replica will be plotted. Output files: **hrex_trajectory_demux.svg** Scatter plot of Hamiltonian visited by each replica in each frame. **hrex_transition_matrix.svg** Empirical transition matrix calculated from the accepted RE moves. **hrex_coord_hamiltonians.svg** Bar plot of the Hamiltonian occupancy of each replica.
convergency	The $\Delta\Delta G_{A\to B}$ and the associated error will be estimated for truncated trajectories at every convergence ps (default: **500**). The same will be done using the reversed trajectory, obtained by reversing the frames. Both $\Delta\Delta G$ for the truncated trajectories ate each time will be plotted and can be used te inspect convergency. Output file: **ddg_vs_time.svg** Forward and reversed trajectories $\Delta\Delta G$ evaluated for different trajectory lengths.
neighbor_ddg	The $\Delta\Delta G$ between adjacent λ windows and its associated errors will be calculated. This will be done reading the *delta_f_* and *d_delta_f_* matrices generated by pymbar. See Alchemlyb documentation for more info. Output file: **ddg_vs_lambda1.svg** Bar plot of estimated free energy difference between each adjacent state.
overlap_matrix	Only available when estimator MBAR is used. The estimated state overlap matrix O_{ij} is an estimate of the probability of observing a sample from state i in state j. See pymbar documentation for further info. Output file: **overlap_matrix.svg** Estimated probability matrix of observing a sample of state i in state j.



λ	0	1	2	3	4	5	6	7	8	9	10	11
0	.98											.02
1		.98				.01					.01	
2			.99							.01		
3				1.00								
4					1.00							
5		.01				.82	.18					
6						.18	.81				.01	
7								1.00				
8									1.00			
9			.01							.99		
10		.01					.01				.97	
11	.02											.98

(a) ddg_vs_time.svg. $\Delta\Delta G$ calculated using trajectories at truncated time steps. This plot can be useful to assess the convergency of the free energy estimative.

(b) **overlap_matrix.svg**. Overlap matrix between FEP states, as calculated by *pymbar*. Each element O_{ij} represents the probability of finding a conformation of the state i in state j. (ADD REF)

Figure 4: Examples of plots generated by PyAutoFEP

4.4 Example plots

PyAutoFEP generates some plots using *alchemlyb-analysis.py* library and some plots using built-in code.

hrex_transition_matrix.svg

4.5 Script options

Option or argument	Accepted values	Comment	Default
input	Valid *.tgz* file or directory	FEP data will be read from this file or directory. If a compressed file is read, it will be uncompressed to a temporary location and processed (See also output_uncompress_directory). A directory structure is expected (**subsection 3.7**).	*No default: required option*
center_molecule	Valid molecule name	Use this molecule as a reference to convert pairwise $\Delta\Delta G$ to $\Delta\Delta G$ in reference to a molecule. A molecule name contained in the perturbation map is expected. By default, if a star or wheel map were used, the center molecule will be automatically detected. If an optimal map is used and this option is provided, network analysis will be performed and $\Delta\Delta G$ to reference will be calculated.	**None**
temperature	Float	The absolute temperature of the sampling. By default, this will be read from the progress file and this option is not required. Use this option to force a different temperature than the read from the progress file.	**None**

Option or argument	**Accepted values**	**Comment**	**Default**
units	**kJmol**, **kcal** or **kBT**	Select unit to be used in the output plots and text.	**kJmol**
first_frame	*int*	First frame (given in ps) to read from the trajectories. By default, frames will be read from the first.	**0**
last_frame	*int*	Last frame (given in ps) to read from the trajectories. By default, frames will be read to the last.	**-1**
convergence	*int* or *list*	Calculates the $\Delta\Delta G$ estimate and error for successive truncated trajectories using this step, in ps. Successive estimates will be plotted to a convergency graph. This can be a step (*int*) or a list of simulation times to use in the convergency analysis (*list*).	**None**, 10-15 evenly-spaced values will be selected by default.
calculate_tau_c	*int*	Analyses each trajectory and calculates the correlation time τ_c using the statistical inefficiency method (ref: 10.1021/acs.jctc.5b00784). This option will not affect the results.	**False**
detect_equilibration	*int*	Automatically (ref: 10.1021/acs.jctc.5b00784) detects a equilibration time in the beginning of each sample and removes the corresponding frames from the analysis.	**False**
estimators	**mbar** or **bar**	Select estimator to estimate $\Delta\Delta G$ (See **subsection 4.1**)	**mbar**
ddg_systems	*list*, currently, the list must contain exactly two items	Select systems X and Y to be used to calculate $\Delta\Delta G$ as $\Delta\Delta G_{A\to B} = \Delta\Delta G_{A\to B_X} - \Delta\Delta G_{A\to B_Y}$. These will be the names of the folders read from each perturbation directory in the data file or dir. Currently, the default value is likely the only useful choice.	[**protein, water**]
analysis_types	*list*	Select which analysis should be executed. Use ”all” to run all available analysis. See **subsection 4.3**.	**all**
center_ddg_method	*String*	A method to calculate the $\Delta\Delta G$ in respect to a reference molecule in the graph. This option only makes sense in conjunction with center_molecule. Valid options are: **shortest**, **shortest_average**, **all_averages**, **all_weighted_averages**. See **subsection 4.2**.	**shortest**
output_uncompress_directory	*String*	By default, if input is a compressed *.tgz* file, *analyze_results.py* will extract data to a temporary location and remove such dir upon completion. By using this option, the uncompressed data will be saved to output_uncompress_directory and kept.	**None**
output_plot_directory	*String*	Save all plots to this directory. By default, plots will be saved to directories in the **$PWD**. If output_uncompress_directory is used, plots will be save to it.	**None**, save to **$PWD**

Option or argument	Accepted values	Comment	Default
output_ddg_to_center	*String*	Save a *.csv* file containing the $\Delta\Delta G$ to the reference molecule. Use **False** to suppress this output. File will be saved to output_plot_directory. See also the related option center_molecule.	**ddg_to_center.csv**

5 Tables

Table 11: λ factors for Van der Waals and Coulomb terms for lambda_input=**lambdas12**

$\lambda =$	**VdW$_{\mathbf{A}}$**	**VdW$_{\mathbf{B}}$**	**Coulomb$_{\mathbf{A}}$**	**Coulomb$_{\mathbf{B}}$**
0	1.00000	0.00000	1.00000	0.00000
1	1.00000	0.11850	0.75000	0.00000
2	1.00000	0.18978	0.50000	0.00000
3	1.00000	0.24741	0.25000	0.00000
4	1.00000	0.32525	0.00000	0.00000
5	0.67479	0.45630	0.00000	0.00000
6	0.45630	0.67479	0.00000	0.00000
7	0.32525	1.00000	0.00000	0.00000
8	0.24741	1.00000	0.00000	0.25000
9	0.18978	1.00000	0.00000	0.50000
10	0.11850	1.00000	0.00000	0.75000
11	0.00000	1.00000	0.00000	1.00000

Table 12: λ factors for Van der Waals and Coulomb terms for lambda_input=**lambdas24**

$\lambda =$	$\mathbf{VdW_A}$	$\mathbf{VdW_B}$	$\mathbf{Coulomb_A}$	$\mathbf{Coulomb_B}$
0	1.00000	0.00000	1.00000	0.00000
1	1.00000	0.00000	0.75000	0.00000
2	1.00000	0.00000	0.50000	0.00000
3	1.00000	0.00000	0.25000	0.00000
4	1.00000	0.00000	0.00000	0.00000
5	0.76285	0.00000	0.00000	0.00000
6	0.57750	0.00000	0.00000	0.00000
7	0.41884	0.00000	0.00000	0.00000
8	0.28687	0.00000	0.00000	0.00000
9	0.18158	0.00000	0.00000	0.00000
10	0.10297	0.00000	0.00000	0.00000
11	0.05106	0.00000	0.00000	0.00000
12	0.00000	0.05106	0.00000	0.00000
13	0.00000	0.10297	0.00000	0.00000
14	0.00000	0.18158	0.00000	0.00000
15	0.00000	0.28687	0.00000	0.00000
16	0.00000	0.41884	0.00000	0.00000
17	0.00000	0.57750	0.00000	0.00000
18	0.00000	0.76285	0.00000	0.00000
19	0.00000	1.00000	0.00000	0.00000
20	0.00000	1.00000	0.00000	0.25000
21	0.00000	1.00000	0.00000	0.50000
22	0.00000	1.00000	0.00000	0.75000
23	0.00000	1.00000	0.00000	1.00000

Table 13: λ factors for Van der Waals and Coulomb terms for lambda_input=**lambdas16wang** (ref:10.1073/pnas.1114017109)

$\lambda =$	$\mathbf{VdW_A}$	$\mathbf{VdW_B}$	$\mathbf{Coulomb_A}$	$\mathbf{Coulomb_B}$
0	1.00000	0.00000	1.00000	0.00000
1	1.00000	0.00000	0.75000	0.00000
2	1.00000	0.00000	0.50000	0.00000
3	1.00000	0.00000	0.25000	0.00000
4	1.00000	0.00000	0.00000	0.00000
5	0.85700	0.14300	0.00000	0.00000
6	0.71400	0.28600	0.00000	0.00000
7	0.57100	0.42900	0.00000	0.00000
8	0.42900	0.57100	0.00000	0.00000
9	0.28600	0.71400	0.00000	0.00000
10	0.14300	0.85700	0.00000	0.00000
11	0.00000	1.00000	0.00000	0.00000
12	0.00000	1.00000	0.00000	0.25000
13	0.00000	1.00000	0.00000	0.50000
14	0.00000	1.00000	0.00000	0.75000
15	0.00000	1.00000	0.00000	1.00000

Table 14: λ factors for Van der Waals and Coulomb terms for lambda_input=**lambdas23wang** (ref:10.1073/pnas.1114017109)

$\lambda =$	$\mathbf{VdW_A}$	$\mathbf{VdW_B}$	$\mathbf{Coulomb_A}$	$\mathbf{Coulomb_B}$
0	1.00000	0.00000	1.00000	0.00000
1	1.00000	0.00000	0.75000	0.00000
2	1.00000	0.00000	0.50000	0.00000
3	1.00000	0.00000	0.25000	0.00000
4	1.00000	0.00000	0.00000	0.00000
5	0.68000	0.00000	0.00000	0.00000
6	0.46000	0.00000	0.00000	0.00000
7	0.33000	0.00000	0.00000	0.00000
8	0.25000	0.00000	0.00000	0.00000
9	0.19000	0.00000	0.00000	0.00000
10	0.12000	0.00000	0.00000	0.00000
11	0.00000	0.00000	0.00000	0.00000
12	0.00000	0.12000	0.00000	0.00000
13	0.00000	0.19000	0.00000	0.00000
14	0.00000	0.25000	0.00000	0.00000
15	0.00000	0.33000	0.00000	0.00000
16	0.00000	0.46000	0.00000	0.00000
17	0.00000	0.68000	0.00000	0.00000
18	0.00000	1.00000	0.00000	0.00000
19	0.00000	1.00000	0.00000	0.25000
20	0.00000	1.00000	0.00000	0.50000
21	0.00000	1.00000	0.00000	0.75000
22	0.00000	1.00000	0.00000	1.00000