# HPC_Drug Documentation

Maurice Karrenbrock

March 24, 2021

# Contents


**1 How to Cite** **1**

**2 User guide** **2**
2.1 Install and Setup . . . . . 2
2.2 Plug and Play . . . . . 4
2.2.1 HREM for FS-DAM Protein-Ligand binding free energy (main.py) . . . . . 4
2.2.2 scripts . . . . . 12
2.3 Python API for Advanced Users . . . . . 13

**3 Developer guide** **17**
3.1 Introduction . . . . . 17
3.2 Contribution Guidelines . . . . . 17
3.3 All Functions and Classes of HPC_Drug (WORK IN PROGRESS) 18
3.4 HPC_Drug/files_IO . . . . . 18
3.4.1 HPC_Drug/files_IO/write_on_files.py . . . . . 18
3.4.2 HPC_Drug/files_IO/read_file.py . . . . . 18
3.5 HPC_Drug/PDB . . . . . 19
3.5.1 HPC_Drug/PDB/biopython.py . . . . . 19
3.5.2 HPC_Drug/PDB/prody.py . . . . . 20
3.5.3 HPC_Drug/PDB/download_pdb.py . . . . . 22
3.5.4 HPC_Drug/PDB/structural_information_and_repair.py 22
3.5.5 HPC_Drug/PDB/remove_trash_metal_ions.py . . . . 23
3.5.6 HPC_Drug/PDB/merge_pdb.py . . . . . 23
3.5.7 HPC_Drug/PDB/remove_disordered_atoms.py . . . . 23
3.5.8 HPC_Drug/PDB/select_model_chain.py . . . . . 24
3.5.9 HPC_Drug/PDB/add_chain_id.py . . . . . 24
3.6 HPC_Drug/PDB/structural_information . . . . . 24
3.6.1 HPC_Drug/PDB/structural_information/mmcif_header.py 24
3.6.2 HPC_Drug/PDB/structural_information/scan_structure.py 27

3.7 HPC_Drug/PDB/repair_pdb . . . . . 30
3.7.1 HPC_Drug/PDB/repair_pdb/repair.py . . . . . 30
3.7.2 HPC_Drug/PDB/repair_pdb/pdbfixer.py . . . . . 30
3.8 HPC_Drug/PDB/organic_ligand . . . . . 31
3.8.1 HPC_Drug/PDB/organic_ligand/primadorac.py . . . 31
3.8.2 HPC_Drug/PDB/organic_ligand/get_ligand_topology.py 32
3.9 HPC_Drug/structures . . . . . 32
3.9.1 HPC_Drug/structures/structure.py . . . . . 33
3.9.2 HPC_Drug/structures/protein.py . . . . . 33
3.9.3 HPC_Drug/structures/ligand.py . . . . . 34
3.9.4 HPC_Drug/structures/get_ligands.py . . . . . 35
3.10 HPC_Drug/auxiliary_functions . . . . . 35
3.10.1 HPC_Drug/auxiliary_functions/path.py . . . . . 35
3.10.2 HPC_Drug/auxiliary_functions/get_iterable.py . . . . 36
3.10.3 HPC_Drug/auxiliary_functions/run.py . . . . . 36

**4 Dependencies** **38**

# Chapter 1

# How to Cite

To cite HPC_Drug please cite the following works:

[1]
Maurice Karrenbrock. HPC_Drug: a python application for Drug Development on High Performance Computing systems. Master's Thesis Università degli Studi di Firenze, Firenze, Italy, June, 2020.

```
@mastersthesis{HPC_Drug_mastersthesis,
author  = "Maurice Karrenbrock",
title   = "{HPC\_Drug: a python application for Drug Development on High Perform
school  = "Università degli Studi di Firenze",
year    = 2020,
address = "Firenze, Italy",
month   = jun
}
```

# Chapter 2

# User guide

## 2.1 Install and Setup

In this section I will give a brief overview on how to install HPC_Drug and how to set up the python environment.

To install the program you simply have to download it from the GitHub repository: https://github.com/MauriceKarrenbrock/HPC_Drug and, if you already had a setup environment, you could already run the main.py program (or if you would like to use one of the other scripts in the scripts/ directory remember to copy it in the root one first).

The environment setup is a little bit longer, in fact HPC_Drug, being a middleware, has a fair amount of dependencies. This numbered list below is one example to get the job done fast and smooth, but you may need to do things differently:

1. download and install miniconda https://docs.conda.io/en/latest/miniconda.html
2. create a conda environment with this command: *conda env create -f environment.yml* the environment.yml file can be found in the root directory and the newly created environment will be called HPC_Drug . As the dependencies of any package can change at any time the environment.yml file might not be up to date, in case install the missing packages manually. Of course if you prefer to have more control over the process you can install everything manually (but it would take some time)
3. activate the environment: *conda activate HPC_Drug*

4. install plumed[2]: if you don't need some of the advanced functionalities of plumed (we won't need them) you already installed it while creating the conda environment (good job!), otherwise you can find all the needed information on the plumed website https://www.plumed.org
5. install primadorac[3] and Orac[4] (they are distributed together): download Orac from it's website http://www.chim.unifi.it/orac and follow the installation guide on the documentation. (As it is a quite challenging task below you will find a little help paragraph for this step)
6. install Gromacs[5]: if you want to use Gromacs instead of Orac as MD program you can install it by following the instructions on the Gromacs website http://www.gromacs.org , in case you have some problems in the compilation process or you need to patch it with plumed (necessary if you want to use a Replica Exchange Method (REM) in older versions, optional in newer versions) I found this blog article very useful https://sajeewasp.com/gromacs-plumed-gpu-linux/ (it is for an old version of Gromacs but still useful).

**Installing Orac and primadorac** Installing Orac and primadorac can be a bit user unfriendly so here is a little installation help, before you start download the gfortran compiler:

1. download and unpack the Orac files (containing the primadorac ones too), we will call this directory orac
2. make a directory called ~/ORAC/trunk (it MUST be in your home)
3. go to orac/src and tipe *./configure -GNU -FFTW* and then type *make*. A new directory called GNU will have appeared
4. if you want to use OpenMP or MPI redo the previous step with the needed flags ex: *./configure -GNU -FFTW -OMP* and you will see more directories being made.
5. copy the orac/lib directory in ~/ORAC/trunk/lib
6. copy any directory you created inside orac/src in ~/ORAC/trunk/src
7. check if the program works (use the executable inside ~/ORAC/trunk/src/GNU*)
8. download the MOPAC2016.exe executable from the openmopac webpage http://openmopac.net and install it correctly
9. go to orac/tools/primadorac and run *make*

10. go to orac/tools/primadorac/www directory and check if there is an executable called new_rms, if not use gfortran to crate it by compiling new_rms.f
11. copy the orac/tools/primadorac directory in ~/ORAC/trunk/tools/primadorac
12. check if primadorac works properly
    (the right executable is ~/ORAC/trunk/tools/primadorac/primadorac.bash)
13. at his point everything should work file

This shall not be taken as a complete Orac and primadorac installation guide but only as an help.

## 2.2 Plug and Play

### 2.2.1 HREM for FS-DAM Protein-Ligand binding free energy (main.py)

The main usage of HPC_Drug program is that of generating the input, that can then be copied in a HPC cluster, for a series of completely independent HREM (Hamiltonian Replica Exchange) MD simulations in order to get the starting configurations for a FS-DAM simulation and the subsequent calculation of the absolute protein-ligand binding free energy. The number of independent HREM that is going to be predisposed is hardware (the HPC cluster architecture) and system (the numbers of atoms in the system) dependent, in fact the goal is to produce 32 ns of simulation in 24 hours wall-time.

At the moment of the writing the input can be done for both Gromacs[5] and Orac[4] MD programs.

The program (HPC_Drug) is able to start from a PDB or mmCIF file given as input, or simply the wwPDB id of the protein that will be downloaded, that contains the organic ligand of interest as an HETATM residue. The program will repair missing atoms and residues, remove useless molecules that are on the PDB (or mmCIF) file only because they were needed to crystallize the protein, produce the needed topology files for the organic ligands (.itp .tpg .prm etc...), rename the residues of the protein in order to be assigned the right force field parameters (usually the ones complexing a metallic ion), fix any bad conformation and wrong atom-atom distance in the structure, find the disulfide bonds, create and optimize a solvent box around

the system and in the end creating the directory to copy on the access node of the HPC cluster in order to start the various independent HREM runs.

All this in a completely automated and independent way, you as a user do only have to create an input file with the needed information and then run this command in the directory you want the data to be stored (if you are interested in the stdout remember to redirect it):

$$\$ \textit{python /path/to/main.py input\_file.txt}$$

### The input file

The input file has a very simple key = value format, some options are compulsory others have a default if omitted. Here is a general overview of the options and below there will be an input example both for Gromacs[5] and Orac[4]. The file is case sensitive and the "#" sign is for comments.

- protein = the protein's wwPDB id (compulsory)
- protein_filetype = the format of the structure file: pdb for PDB files, cif for mmCIF files (default cif)
- Protein_model = protein structures may have various models, this is the one that will be chosen, it starts from 0 (zero) and the default is 0
- Protein_chain = many proteins are made of more than one polypeptidic chain, but as the calculation of protein-ligand binding free energy does have only sense when there is one ligand and one chain this is the PDB chain id that you want to work on. Default A
- ph = the ph at which the hydrogens shall be added to the protein default 7.0
- repairing_method = the tool with which the pdb shall be fixed by adding missing atoms, missing residues, missing hydrogens and substituting non standard residues with standard ones the default is pdbfixer[6] (needs openmm[7])
- local = local tells if the program shall use a protein file that is already on the computer if 'no' (default) it will if 'yes' insert the absolute path in filepath download it from the wwPDB database
- filepath = the path to the PDB (or mmCIF) file if local = yes, using the absolute path is more robust

- ligand_in_protein = if yes the program will check for the ligand inside the given protein file, if no the ligand must be given as a separated file (not implemented yet), default yes
- ligand = it is the (absolute) path to the pdb file of the ligand if ligand_in_protein = yes
- ligand_elaboration_program = The program with which elaborate the ligand (optimization and force field), default primadorac (amber force field)
- ligand_elaboration_program_path = the (absolute) path to the ligand_elaboration_program executable
- MD_program = The molecular dynamics MD program of choice, default gromacs, a working executable of the program must be present on your PC
- MD_program_path = the (absolute) path to the MD program executable
- protein_prm_file = if MD_program = it is the orac .prm file for the protein
- protein_tpg_file = for gromacs it is the force field to use for the protein (more information in the gromacs example), for orac it is the .tpg file for the protein
- solvent_pdb = if MD_program = gromacs it is the model used for the solvent molecules (more information in the gromacs example), if orac it is the pdb of one solvent molecule
- residue_substitution = how to rename the metal binding residues, standard (default) or custom_zinc[8]
- kind_of_processor = the kind of processor that is present on the HPC cluster, default skylake, other options broadwell knl (you can find them in the important_lists.py file)
- number_of_cores_per_node = how many cores there are on the HPC cluster on each node, default 64
- gpu_per_node = the number of GPUs per node, default 1, if GPUs shall not be used will be ignored
- number_of_hrem_replicas_per_battery_bound = the number of HREM replicas per each Battery bound state, default 8

- number_of_hrem_replicas_per_battery_unbound = the number of HREM replicas per each Battery unbound state, default 8
- bound_batteries = number of bound batteries per HREM, default auto
- unbound_batteries = number of unbound batteries per HREM, default auto
- n_steps_bound = number of MD steps (reference state) for bound HREM, dafault auto
- n_steps_unbound = number of MD steps (reference state) for unbound HREM, dafault auto
- timestep_bound = timestep for bound HREM, dafault auto
- timestep_unbound = timestep for unbound HREM, dafault auto

### Orac example

This is an example of correct input with some explanatory comments (below it you will find more information):

```
#This is a correct input example
#Every line beginning with '#' is a comment
#Error occurs only for wrong keys
#case-sensitive
#-----------------------------------------------

#Protein code and desired file type, possible values 'cif' or 'pdb'
#'cif' is for PDBx/mmCIF (default)
#'pdb' for standard PDB file (not implemented)

protein = 1df8
#3m5e
protein_filetype = cif

#the model to take from the mmcif, if omitted model = 0 will be taken (starts fr
#Protein_model = 0
#the chain to choose from the xray structure (default A)
#Protein_chain = A

#The ph at which the hydrogens shall be added to the protein
#default = 7.0
```

```
ph = 7.0

#with which tool the pdb shall be fixed
#adding missing atoms, missing hydrogens and substituting
#non standard residues with standard ones
#default is pdbfixer (needs openmm and conda environment)

repairing_method = pdbfixer

#local tells if the program shall use a protein file that is already
#on the computer
#if 'no' (default) it will download it from the wwPDB database
#if 'yes' insert the absolute path in filepath
#any keyword different from yes and no will abort the program

local = no
#filepath = 2rfh.pdb

#if ligand_in_protein = yes (default) ligand will be taken from the
#protein mmcif file

#if ligand_in_protein = no ligand will be given as input
#ligand = path/to/PDB
#the ligand shall be given as a pdb file (I suggest to use absolute path)

ligand_in_protein = yes
#ligand = ligand.pdb

#The program with which elaborate the ligand (optimization and potential)
#default primadorac (amber force field)

ligand_elaboration_program = primadorac
ligand_elaboration_program_path = ~/ORAC/trunk/tools/primadorac/primadorac.bash


#The molecular dynamics MD program
#of choice and the path to the executable

MD_program = orac
MD_program_path = ~/ORAC/trunk/src/GNU-FFTW-OMP/orac
```

```
#protein tpg and primadorac
#if omitted default ones will be used
#see HPC_Drug/lib/
protein_prm_file = amber99sb-ildn.prm
protein_tpg_file = amber99sb-ildn.tpg

#solvent pdb, if omitted default will be used
#see HPC_Drug/lib/
solvent_pdb = water.pdb

#there can be custom residue substitutions for metal binding residues
#default standard
#standard, custom_zinc
residue_substitution = standard

#The kind of processor present on the HPC cluster
#default skylake
#other options broadwell knl (you can find them in the important_lists.py file)
kind_of_processor = skylake

# how many cores there are on the HPC cluster on each node
#default 64
number_of_cores_per_node = 64

#number of HREM replicas per each Battery, default 8
number_of_hrem_replicas_per_battery = 8
```

In the end you will obtain a directory called {protein id}_REM that can be copied on the access node of the HPC cluster you want to use. It doesn't only contain the input file for Orac[4] but also some basic PBS and SLURM input files to run the code with the right amount of processors, but pay attention, these are very basic so you will 99.9% need to add/edit some lines.

### Gromacs example

This is an example of correct input with some explanatory comments (below it you will find more information):

```
#This is a correct input example
#Every line beginning with '#' is a comment
#Error occurs only for wrong keys
#case-sensitive
```

```
#-----------------------------------------------

#Protein code and desired file type, possible values 'cif' or 'pdb'
#'cif' is for PDBx/mmCIF (default)
#'pdb' for standard PDB file (not implemented)

protein = 1df8
#3m5e
protein_filetype = cif

#the model to take from the mmcif, if omitted model = 0 will be taken (starts fr
#Protein_model = 0
#the chain to choose from the xray structure (default A)
#Protein_chain = A

#The ph at which the hydrogens shall be added to the protein
#default = 7.0
ph = 7.0

#with which tool the pdb shall be fixed
#adding missing atoms, missing hydrogens and substituting
#non standard residues with standard ones
#default is pdbfixer (needs openmm and conda environment)

repairing_method = pdbfixer

#local tells if the program shall use a protein file that is already
#on the computer
#if 'no' (default) it will download it from the wwPDB database
#if 'yes' insert the absolute path in filepath
#any keyword different from yes and no will abort the program

local = no
#filepath = 2rfh.pdb

#if ligand_in_protein = yes (default) ligand will be taken from the
#protein mmcif file

#if ligand_in_protein = no ligand will be given as input
#ligand = path/to/PDB
#the ligand shall be given as a pdb file (I suggest to use absolute path)
```

```
ligand_in_protein = yes
#ligand = ligand.pdb

#The program with which elaborate the ligand (optimization and potential)
#default primadorac (amber force field)

ligand_elaboration_program = primadorac
ligand_elaboration_program_path = ~/ORAC/trunk/tools/primadorac/primadorac.bash


#The molecular dynamics MD program
#of choice and the path to the executable

MD_program = orac
MD_program_path = ~/ORAC/trunk/src/GNU-FFTW-OMP/orac

#protein tpg and primadorac
#if omitted default ones will be used
#see HPC_Drug/lib/
protein_prm_file = amber99sb-ildn.prm
protein_tpg_file = amber99sb-ildn.tpg

#solvent pdb, if omitted default will be used
#see HPC_Drug/lib/
solvent_pdb = water.pdb

#there can be custom residue substitutions for metal binding residues
#default standard
#standard, custom_zinc
residue_substitution = standard

#The kind of processor present on the HPC cluster
#default skylake
#other options broadwell knl (you can find them in the important_lists.py file)
kind_of_processor = skylake

# how many cores there are on the HPC cluster on each node
#default 64
number_of_cores_per_node = 64
```

```
#number of HREM replicas per each Battery, default 8
number_of_hrem_replicas_per_battery = 8
```

In the end you will obtain two directories, one to use if you have a Gromacs[5] patched with Plumed[2] on your HPC cluster of choice and the other to use if you want to use Gromacs' native Replica Exchange (what we do is actually trick it to think we are doing a temperature REM), that can be copied on the access node of the HPC cluster you want to use (but for both versions you will need a working Plumed executable on your PC, the one you can download from conda-forge is perfect). They do not only contain the input files for Gromacs but also some basic PBS and SLURM input files to run the code with the right amount of processors, but pay attention, these are very basic so you will 99.9% need to add/edit some lines, and a bash script to create all the needed .tpr files once you are on the HPC cluster (must be run before the workload-manager input).

### 2.2.2 scripts

In the scripts directory can be found other possible uses of the HPC_Drug classes and functions. This secondary programs get things done like automatizing the main.py process on many proteins, repairing a given protein and separating the protein from the ligand pdb etc...

#### automated_main.py

This program does the same thing as main.py, but on a list of protein ids and creates a different directory for each (named after the protein id), the input file must contain a protein id for each line, like:
1dz8
2gz7
3sn8
etc...

And the command is:

$$\$\ python\ automizable\_main.py\ input\_file.txt$$

Of course you must check for the other options inside the .py file. stderr and stdout are redirected to two different files for any given protein.

## 2.3 Python API for Advanced Users

In this section I will show the usage of some functions and classes that a common user could find useful for the development of custom pipelines. To get a more detailed knowledge of all the classes and functions of HPC_Drug checkout the developer guide section.

### Class GetProteinLigandFilesPipeline(Pipeline)

It is a subclass of the Pipeline class. Here is an example of instantiation:

*f*rom HPC_Drug import pipelines

my_object = pipelines.GetProteinLigandFilesPipeline(
protein = '2gz7',
protein_filetype = 'cif',
local = 'no',
filepath = None,
ligand = None,
Protein_model = 0,
Protein_chain = 'A',
repairing_method = 'pdbfixer')

protein is compulsory, and if local = 'yes' (meaning that the protein file is already on your PC and shall not be downloaded from the wwPDB) filepath must be given as a string. For any other option the default is the one written above.

The only public method is execute() and returns a HPC_Drug.structures.protein.Protein instance containing a repaired PDB file of the protein with its metallic ions, and a list of HPC_Drug.structures.ligand.Ligand instances for any organic (not trash) ligand found in the structure. The protein PDB will only contain the selected Protein_model model and Protein_chain chain.

### Class Structure(object)

It is the superclass for all the structure classes (like Protein and Ligand), it's not instantiatable (it's contructor raises a NotImplementedError), but implements some common methods that subclasses will inherit.

**Method write(file_name = None, struct_type = 'biopython')** Writes the self.structure structure on the file_name file and will update self.pdb_file with the new name (if it is omitted will overwrite the existing self.pdb_file),

struct_type tells the function with which tool self.structure was obtained (biopython, prody) the default is "biopython"

**update_structure(struct_type = "biopython")** Updates self.structure parsing self.pdb_file, struct_type is the tool you want to use (biopython, prody) default "biopython"

### Class Protein(HPC_Drug.structures.structure.Structure)

It is the Protein class, one of the fundamental classes of the program, it contains any possible information about the protein you are studying, it subclasses the HPC_Drug.structures.structure.Structure class adding some methods to it, and overwriting the ___init___ method:

- protein_id = the protein wwPDB id (string)
- pdb_file = the PDB or mmCIF file of the protein, default {protein_id}.{file_type}
- structure = the Biopython[9] or the Prody[10] structure parsed from the pdb_file
- substitutions_dict = a dictionary that contains information about the metal binding residues and the cysteines that make a disulf bond
- sulf_bonds = a list of tuples containing the couples of cysteines binding in a disulf bond
- seqres = a place where to store the residue sequence if needed
- file_type = can be 'cif' or 'pdb' depending on the protein_pdb file format (mmCIF or PDB), default 'cif'
- model = integer, the model taken in consideration (starts from 0), default 0
- chain = string, the PDB chain taken in consideration, default 'A'
- gro_file = the Gromacs[5] .gro file
- top_file = the Gromacs[5] .top file
- tpg_file the .tpg file, needed for Orac[4]
- prm_file the .prm file, needed for Orac[4]
- _ligands = the organic ligands, it is private but there is a method to get them (see below)

This is an example instantiation:
*from HPC_Drug.structures import protein*

*my_protein = protein.Protein(protein_id = "2gz7", pdb_file = "2gz7.cif", file_type = "cif", model = 0, chain = "A")*

Besides the superclass methods Protein implements the above methods:

**Method add_ligands(Ligand)** Takes a HPC_Drug.structures.ligand.Ligand instance and add it to self._ligands

**Method clear_ligands()** Clears ALL the ligands stored in self._ligands

**Method update_ligands(ligands)** Takes an iterable (list, tuple, etc...) containing HPC_Drug.structures.ligand.Ligand instances and ovewrites self._ligand with this new ones (any information about the old ones will be lost)

ligands :: iterable containing the new HPC_Drug.structures.ligand.Ligand instances

**Method get_ligand_list()** Returns a list with the pointers to self._ligands (it is not a copy of them so pay attention on what you do)

**Class Ligand(HPC_Drug.structures.structure.Structure)**

It is the Ligand class, one of the fundamental classes of the program, it contains any possible information about an organic ligand of the studied protein, it subclasses the HPC_Drug.structures.structure.Structure class only overwriting the ___init___ method:

- resname = the ligand residue name (string) capital letters
- pdb_file = the PDB or mmCIF file of the ligand, default {resname}.{file_type}
- structure = the Biopython[9] or the Prody[10] structure parsed from the pdb_file
- resnum = integer, the residue number inside the original PDB (or mmCIF) file from which the ligand was or will be extracted ( it is very useful if the ligand has not been extracted yet)
- file_type = can be 'cif' or 'pdb' depending on the protein_pdb file format (mmCIF or PDB), default 'pdb'

- itp_file = the Gromacs[5] .itp file
- gro_file = the Gromacs[5] .gro file
- top_file = the Gromacs[5] .top file
- tpg_file the .tpg file, needed for Orac[4]
- prm_file the .prm file, needed for Orac[4]

This is an example instantiation:
*from HPC_Drug.structures import ligand*

*my_ligand = ligand.Ligand(resname = "LIG", pdb_file = "LIG.pdb", file_type = "pdb", resnum = 4)*

# Chapter 3

# Developer guide

## 3.1 Introduction

The first part is a little guideline for who would like to contribute with at the open-source HPC_Drug project, and the second is a list of all the functions and classes of the program with a brief description (they actually are the copy paste of the comments in the code).

## 3.2 Contribution Guidelines

If you would like to contribute to the project (it is an open-source software licensed with the agpl v3 license) can do it though the GitHub repository https://github.com/MauriceKarrenbrock/HPC_Drug
If you found a bug, or have an idea for an improvement simply open an issue. If you would like to contribute with some code first open an issue in order to talk about your idea and hear the opinion of the other users and developers, it would be a bit of a pity if you did a lot of work on something no one agrees on. If it is a little thing link your pull request to the issue from the beginning in order to check if everything is ok, or if there might be needed some changes to accept it. If yours is a bigger idea please write WIP in the end of the issue title so everyone knows it is a work in progress with no available pull request yet or that could need more than one pull request , in this case the issue would be used as a place to discuss and talk about the implementation of the new idea, to solve problems, and to answer any possible question.

If you are writing a piece of code please remember to make it as modular, flexible, maintainable, readable and pythonic as possible, speed in fact is not

a goal of this program, but expandability and flexibility are. And please try to write the needed unit-tests (and if possible integration and end to end tests), try to comment (and write) the code in a way that future developers will be able to understand it, and, if it makes sense for the kind of contribution you made, update the documentation with your new creation.

For the rest have fun and may the Force of Drug Discovery be with you!

## 3.3 All Functions and Classes of HPC_Drug (WORK IN PROGRESS)

## 3.4 HPC_Drug/files_IO

### 3.4.1 HPC_Drug/files_IO/write_on_files.py

**Function write_file(lines, file_name = "file.txt")**

This function writes a new file or overrides an existing one (no safety check is done!). lines can be a single string or an iterable (list, tuple etc...) and contains the lines that will be written on the file. file_name must be a string and is the name of the file that will be created.
Absolutely no formatting is done on the strings so they must be already formatted properly (like with newline
$n$).

**Function append_file(lines, file_name = "file.txt")**

This function appends some lines to an existing file. lines can be a single string or an iterable (list, tuple etc...) and contains the lines that will be written on the file. file_name must be a string and is the name of the file that will be edited.
Absolutely no formatting is done on the strings so they must be already formatted properly (like with newline
$n$).

### 3.4.2 HPC_Drug/files_IO/read_file.py

**Function read_file(file_name)**

Reads a file and returns a list containing the lines of the file.
Can be resource consuming on very large files.

file_name must be a string

## 3.5 HPC_Drug/PDB

This folder contains some functions to parse and write PDB and mmCIF files with multiple tools (Biopython, Prody).

### 3.5.1 HPC_Drug/PDB/biopython.py

**Function parse_pdb(protein_id, file_name)**

This function uses Biopython Bio.PDB.PDBParser to return a Biopython structure from a PDB file.

protein_id :: string file_name :: string

return structure

**Function parse_mmcif(protein_id, file_name)**

This function uses Biopython Bio.PDB.MMCIFParser to return a Biopython structure from a mmCIF file.

protein_id :: string file_name :: string

return structure

**Function structure_factory(Protein)**

This is a function that uses the right parse_ function depending on Protein.file_type

Protein :: HPC_Drug.structures.Protein instance or HPC_Drug.structures.Ligand instance or wathever has a file_type and pdb_file attribute

return structure

**Function mmcif2dict(file_name)**

Uses Bio.PDB.MMCIF2DICT to return a dictionary of the mmcif file

return Bio.PDB.MMCIF2Dict.MMCIF2Dict(file_name)

**Function write_pdb(structure, file_name = "file.pdb")**

writes a pdb file when given a Biopython structure

structure :: is instance Bio.PDB.Entity.Entity

file_name :: string, default file.pdb

returns nothing

**Function write_mmcif(structure, file_name = "file.cif")**

writes a mmCIF file when given a Biopython structure

structure :: is instance Bio.PDB.Entity.Entity

file_name :: string, default file.cif

returns nothing

**Function write_dict2mmcif(dictionary, file_name = "file.cif")**

Writes a mmcif file starting from a dictionary obtained from mmcif2dict (that uses Bio.PDB.MMCIF2DICT )

dictionary :: a dictionary containing all the mmcif infos, obtained with mmcif2dict

returns nothing

**Function write(structure, file_type = "pdb", file_name = None)**

This is a factory that writes the file given a structure or a mmcif2dict dictionary, the file type (pdb mmcif) and the output file name

structure :: a Bio.PDB.Entity instance or a mmcif2dict dictionary, if you give a dictionary only file_type = 'cif' will be accepted

file_type :: string, pdb or cif, default pdb

file_name :: string, default file.file_type

returns nothing

### 3.5.2 HPC_Drug/PDB/prody.py

**Function parse_pdb(file_name)**

Parses a PDB file with ProDy and returns a ProDy structure (prody.AtomGroup)

file_name :: string

returns structure

**Function write_pdb(structure, file_name = "file.pdb")**

Takes a Prody strucure prody.AtomGroup and writes it on a pdb file

strucure :: prody.AtomGroup

file_name :: string, default "file.pdb"

returns nothing

**Function select(structure, string)**

Uses the Prody select function with string as command

structure :: prody.AtomGroup

string :: string, this is the command that will be passed to prody select

returns a new prody.AtomGroup

**Class ProdySelect(object)**

This cass is a smart facade that implements some useful uses of HPC_Drug.PDB.prody.select

**Method ___init___(self, structure)** The strucure must be a prody.AtomGroup

**Method only_protein(self)** Returns a prody structure containing only the protein

**Method protein_and_ions(self)** Returns a prody structure containing only the protein and the inorganic ions

**Method resname(self, resname)** Given a resname returns a Prody structure only containing any residue with that residue name

resname :: string

**Method resnum(self, resnum)** Given a resnum returns a Prody structure only containing the residue with that residue number

resnum :: integer

### 3.5.3 HPC_Drug/PDB/download_pdb.py

**Function download(protein_id, file_type = 'cif', pdir = None)**

The function downloads a PDB or a mmCIF from wwwPDB in a selected directory the default directory is the working directory it returns the filename (str)

protein_id :: string, it is the protein to download

file_type :: string, it can be pdb or cif depending on the format required, default cif

pdir :: string, default working directory, the directory where the file is saved

return file_name , string

raises a FileNotFoundError if the file is not downloaded correctly

### 3.5.4 HPC_Drug/PDB/structural_information_and_repair.py

This file contains a template to get the residues near a metallic ion, disulf bonds and organic ligand's renames and resnumbers from a Protein instance and repair the PDB (or mmCIF) file

**Class InfoRepair(object)**

**Method ___init___(self, Protein, repairing_method = "pdbfixer")**
Constructor

**Method _parse_header(self)** private

**Method _parse_structure(self)** private

**Method _repair(self)** private

**Method _pdb(self)** private

**Method _cif(self)** private

**Method get_info_and_repair(self)** Returns Protein and organic_ligand_list

Protein.pdb_file is a repaired PDB or mmCIF file

return Protein, organic_ligand_list

### 3.5.5 HPC_Drug/PDB/remove_trash_metal_ions.py

**Function remove_trash_metal_ions(Protein, trash = important_lists.trash_ions)**

This function removes unwanted metal ions that are still inside the structure after it went through prody selection (updates Protein.pdb_file)

This is a brutal function I will need to do a better job

Protein :: HPC_Drug.structures.protein.Protein instance

Protein.file_type must be pdb or cif otherwise TypeError will be raised

return Protein

### 3.5.6 HPC_Drug/PDB/merge_pdb.py

This file contains the functions to merge PDB or mmCIF. They are useful wen you need to merge one or more organic ligands with a protein.

**Function merge_pdb(Protein)**

Will put all the given ligands after the protein and update the ligand resnums this function is brutal and memory consuming I should do it better in the future

both th protein and the ligands should be in PDB files (no check will be done)

Protein :: HPC_Drug.structures.protein.Protein instance with a valid _ligands value

return Protein with updated Protein.pdb_file

### 3.5.7 HPC_Drug/PDB/remove_disordered_atoms.py

**Function remove_disordered_atoms(Protein)**

Removes disordered atoms, solves a problem about "copied atoms don't inherit disordered_get_list in Biopython"

Protein :: HPC_Drug.structures.protein.Protein instance

return Protein

### 3.5.8 HPC_Drug/PDB/select_model_chain.py

**Function select_model_chain(Protein)**

Takes a Protein instance containing the filename of a PDB or a mmcif Returns a Protein instance with an updated pdb or mmcif file using biopython selects only a chosen model and chain

Protein.chain must be a string Protein.model must be an integer

Protein :: HPC_Drug.structures.protein.Protein instance

return Protein

### 3.5.9 HPC_Drug/PDB/add_chain_id.py

**Function add_chain_id(pdb_file, chain = "A")**

This is a patch because orac and primadorac remove the chain id from pdb files and this confuses some pdb parsers (works on PDB files only)

pdb_file :: string, the pdb file to edit

chain :: string, default A, the chain id to add to the pdb_file

returns nothing

## 3.6 HPC_Drug/PDB/structural_information

### 3.6.1 HPC_Drug/PDB/structural_information/mmcif_header.py

This file contains the files necessary to parse the header of a mmCIF file

**Function get_ligand_binding_residues(mmcif2dict, metals = important_lists.metals)**

This function is called from get_metalbinding_disulf_ligands

Searces the given mmcif file for the metal binding residues parsing the header returns a dictionary that has as key the residue number and as vaue a tuple with (resname, binding atom, metal) for metal binding residues

mmcif2dict :: a dictionary of the type you obtain with HPC_Drug.PDB.biopython.mmcif2dict function

metals :: a list (or tuple etc) that contains all the resnames (in capital letters) of metals necessary to look for, default HPC_Drug.important_lists.metals (Actually the easiest way to personalize metals is to append your custom values to this list)

return resnum : (resname, binding atom, metal), ...

**Function get_disulf_bonds(mmcif2dict)**

This function is called from get_metalbinding_disulf_ligands

Searces the given mmcif file for disulf bonds parsing the header returns a dictionary that has as key the residue number and as vaue a tuple with ('CYS', 'SG', 'disulf') for any disulf cysteine.
And a a list composed of tuples containing the resnumbers of the 2 CYS that bound through disulfide bond

mmcif2dict :: a dictionary of the type you obtain with HPC_Drug.PDB.biopython.mmcif2dict function

return resnum : ('CYS', 'SG', 'disulf'), ... [(resnum, resnum), (...), ...]

**Function get_organic_ligands(mmcif2dict, protein_chain = None, trash = important_lists.trash, metals = important_lists.metals)**

This function is called from get_metalbinding_disulf_ligands

Searces the given mmcif file for organic ligands parsing the header returns a list of resnames. If protein chain is None (default) will list all ligands from any chain, if protein chain is set does only consider the ones of the given chain (es A)

mmcif2dict :: a dictionary of the type you obtain with HPC_Drug.PDB.biopython.mmcif2dict function

protein_chain :: string, default None, the chain id of the chain you want to analize in capital letters (es A)

trash :: a list (or tuple etc) that contains all the resnames (in capital letters) of trash ligands to avoid listing, default HPC_Drug.important_lists.trash (Actually the easiest way to personalize trash is to append your custom values to this list)

metals :: a list (or tuple etc) that contains all the resnames (in capital letters) of metals necessary to look for, default HPC_Drug.important_lists.metals (Actually the easiest way to personalize metals is to append your custom values to this list)

return [resname, resname, ...]

**Function get_ligand_resnum(structure, ligand_resnames = None, protein_chain = 'A', protein_model = 0)**

This function is called from get_metalbinding_disulf_ligands

Given a Biopython structure and a list of Ligand_resnames will return a list containing the ligand resnames and resnumbers in order to distinguish ligands with the same resname: [[resname, resnumber], [.., ...], ...]

ligand_resnames :: list, it is a list containing the organic ligand resnames (capital letters) to look for if it is == None or empty will return None

protein_chain :: string, default A, the chain id of the chain you want to analize in capital letters (es A), if == None no chain selection will be done

protein_model :: integer, default 0, the model to check, if == None no chain and no model selection will be done

return [[resname, resnumber], [.., ...], ...]

**Function get_metalbinding_disulf_ligands(Protein, trash = important_lists.trash, metals = important_lists.metals)**

This is a template that uses the other funcions on this file to return a dictionary with key = resnum and value = (resname, binding atom, metal) or ('CYS', 'SG', 'disulf') depending if the residue number resnum binds a metallic ion or is part of a disulf bond and updates Protein.substitutions_dict with it

and a list of tuples that contain the couples of CYS that are part of a disulf bond and updates Protein.sulf_bonds with it

and a list of tuples with the residue name and residue number of the organic ligands (if there are none None will be returned)

Protein :: a HPC_Drug.structures.protein.Protein instance

trash :: a list (or tuple etc) that contains all the resnames (in capital letters) of trash ligands to avoid listing, default HPC_Drug.important_lists.trash

(Actually the easiest way to personalize trash is to append your custom values to this list)

metals :: a list (or tuple etc) that contains all the resnames (in capital letters) of metals necessary to look for, default HPC_Drug.important_lists.metals (Actually the easiest way to personalize metals is to append your custom values to this list)

return Protein, [[lig_resname, lig_resnum], ...]

### 3.6.2 HPC_Drug/PDB/structural_information/scan_structure.py

This file contains the functions necessary to scan the structure of a PDB file or a headerless mmCIF file

**Function get_metal_binding_residues_with_no_header(structure, cutoff = 3.0, protein_chain = 'A', protein_model = 0, COM_distance = 10.0, metals = important_lists.metals)**

This function gets called by get_metalbinding_disulf_ligands

This function iterates through the structure many times in order to return the metal binding residues through a substitution dictionary

residue_id : [residue_name, binding_atom, binding_metal]

It uses biopython structures

structure :: a biopython structure of the protein

cutoff :: double the maximum distance that a residue's center of mass and a metal ion can have to be considered binding default 3.0 angstrom

protein_chain :: string default 'A', if == None no chain selection will be done

protein_model :: integer default 0, if == None no model and no chain selection will be done

metals :: a list (or tuple etc) that contains all the resnames (in capital letters) of metals necessary to look for, default HPC_Drug.important_lists.metals (Actually the easiest way to personalize metals is to append your custom values to this list)

this function is slow and error prone and should only be used if there is no mmCIF with a good header

It should not be necessary to change COM_distance because it simply is the distance between the center of mass of a residue and the metal that is used to know which atom distances to calculate

**Function get_disulf_bonds_with_no_header(structure, cutoff = 3.0, protein_chain = 'A', protein_model = 0)**

This function gets called by get_metalbinding_disulf_ligands

This function iterates through the structure many times in order to return the disulf bonds through a substitution dictionary and a list of the binded couples

residue_id : [residue_name, binding_atom, binding_metal] and [(cys_id, cys_id), (cys_id, ...), ...]

return substitutions_dict, sulf_bonds

it uses a biopython structure

structure :: biopython structure of the protein

cutoff :: double the maximum distance that two CYS S atoms can have to be considered binding default 3.0 angstrom

protein_chain :: string default 'A', if == None no chain selection will be done

protein_model :: integer default 0, if == None no model and no chain selection will be done

this function is slow and error prone and should only be used if there is no mmCIF with a good header

**Function get_organic_ligands_with_no_header(structure, protein_chain = 'A', protein_model = 0, trash = important_lists.trash, metals = important_lists.metals)**

This function gets called by get_metalbinding_disulf_ligands

This function iterates through the structure to get the organic ligand

returning a list of lists containing [[resname, resnumber], [resname, resnumber], ...]

If there are none returns None

it uses a biopython structure

structure :: biopython structure of the protein

protein_chain :: string default 'A', if == None no chain selection will be done

protein_model :: integer default 0, if == None no model and no chain selection will be done

trash :: a list (or tuple etc) that contains all the resnames (in capital letters) of trash ligands to avoid listing, default HPC_Drug.important_lists.trash (Actually the easiest way to personalize trash is to append your custom values to this list)

metals :: a list (or tuple etc) that contains all the resnames (in capital letters) of metals necessary to look for, default HPC_Drug.important_lists.metals (Actually the easiest way to personalize metals is to append your custom values to this list)

this function is slow and error prone and should only be used if there is no mmCIF with a good header

**Function et_metalbinding_disulf_ligands(Protein, trash = important_lists.trash, metals = important_lists.metals)**

This is a template that uses the other funcions on this file to return a dictionary with key = resnum and value = (resname, binding atom, metal) or ('CYS', 'SG', 'disulf') depending if the residue number resnum binds a metallic ion or is part of a disulf bond and updates Protein.substitutions_dict with it

and a list of tuples that contain the couples of CYS that are part of a disulf bond and updates Protein.sulf_bonds with it

and a list of tuples with the residue name and residue number of the organic ligands (if there are none None will be returned)

Protein :: a HPC_Drug.structures.protein.Protein instance

trash :: a list (or tuple etc) that contains all the resnames (in capital letters) of trash ligands to avoid listing, default HPC_Drug.important_lists.trash (Actually the easiest way to personalize trash is to append your custom values to this list)

metals :: a list (or tuple etc) that contains all the resnames (in capital letters) of metals necessary to look for, default HPC_Drug.important_lists.metals

(Actually the easiest way to personalize metals is to append your custom values to this list)

return Protein, [[lig_resname, lig_resnum], ...]

## 3.7 HPC_Drug/PDB/repair_pdb

### 3.7.1 HPC_Drug/PDB/repair_pdb/repair.py

This file contains the function that repairs a pdb or mmcif file using the right repairing method

**Function repair(Protein, repairing_method = "pdbfixer")**

This is a factory that returns a Protein with Protein.pdb_file updated to a repaired pdb or mmcif file using the right repiring method

Protein :: HPC_Drug.structures.protein.Protein instance

repairing_method :: string, default pdbfixer, it is the tool you want to repair the file with if you input a non existing tool will return NotImplementedError

### 3.7.2 HPC_Drug/PDB/repair_pdb/pdbfixer.py

This file contains the function to repair a PDB or a mmCIF file with pdbfixer

**Function _repair(input_file_name, file_type, output_file_name, add_H = False, ph = 7.0)**

Private, it is called by the repair function

repairs a PDB or mmCIF file with pdbfixer and returns the new file_name

input_file_name :: string, the pdb or mmcif file to be repaired

file_type :: string, can be cif or pdb

output_file_name :: string, the name of the new structure file that will be created

add_H :: bool, default False, if True pdbfixer will add hydrogens according to ph

ph :: float, default 7.0, if add_H == True this is the pH value that will be used to add hydrogens

**Function repair(Protein)**

repairs a PDB or mmCIF file with pdbfixer and returns the new file_name

This function calls _repair it is an interface to use a HPC_Drug.structures.protein.Protein instance on _repair in a simplified way

Protein :: HPC_Drug.structures.protein.Protein instance

return Protein

## 3.8 HPC_Drug/PDB/organic_ligand

### 3.8.1 HPC_Drug/PDB/organic_ligand/primadorac.py

This file contains the class to run primadorac

**Class Primadorac(object)**

It is a template class to run primadorac

**Method ___init___(self, Protein, primadorac_path, ph = 7.0)** Protein :: Protein :: HPC_Drug.structures.protein.Protein instance with a valid Protein._ligands (Protein.get_ligand_list())

primadorac_path :: string, the path to the primadorac executable (better if absolute)

ph :: float, default 7.0, the ph at which the ligand will be protonated (at the moment primadorac does ONLY SUPPORT PH 7.0)

**Method _run(self, string)** private

**Method _rename_itp(self, file_to_search, ligand_resname = "LIG")** private

it is a patch because some old versions of primadorac do mess up the .itp file name

**Method _edit_itp(self, ligand_resname, itp_file)** private

primadorac itp call any lignd LIG i change it to the ligand_resname

and removes the first 9 lines of the file (they make gromacs fail)

**Method execute(self)** Run primadorac

return Protein

with updated .itp .prm .tpg files for any ligand in Protein._ligands

### 3.8.2 HPC_Drug/PDB/organic_ligand/get_ligand_topology.py

This file contains the function that uses the right tool to get the topology of a given organic ligand (.itp .tpg .prm etc...) in order to use it in a MD run

**Function get_topology(Protein, program_path, tool = "primadorac", ph = 7.0)**

Uses the right tool to get the topology of a given organic ligand (.itp .tpg .prm etc...) in order to use it in a MD run returns the updated protein

Protein :: HPC_Drug.structures.protein.Protein instance with a valid Protein._ligands (Protein.get_ligand_list()) value (if it is None or [] will return the protein untouched)

program_path :: string, the absolute path to the tool's executable

tool :: string, default primadorac, the tool to use to get the topology (.itp .tpg .prm etc...)

ph :: float, default 7.0, the ph at which the ligand shall be added the missing hydrogens

return Protein

## 3.9 HPC_Drug/structures

This folder contains the structure classes (Protein, Ligand, Structure)

### 3.9.1 HPC_Drug/structures/structure.py

**Class Ligand(object)**

This is the super class for all structures, it's constructor raises a NotImplementedError.
It implements some common methods.

**Method ___init___(self)** Raises NotImplementedError

**Method write(self, file_name = None, struct_type = 'biopython')** This method writes self.structure on a self.file_type file (pdb, cif) using biopython (default) or prody (can only write pdb files)
If no file_name is given self.pdb_file file will be overwritten otherwise a new file called file_name will be created and self.pdb_file will be updated with the new file_name

file_name :: string, default self.pdb_file

struct_type :: string, values: biopython (default), prody ; is the kind of structure in self.structure

**Method update_structure(self, struct_type = "biopython")** Parses the self.pdb_file file with the selected tool (biopython (default), mmcif2dict or prody) and updates self.structure
prody can only parse pdb files, if you try to parse a cif with prody a TypeError will be raised
mmcif2dict can only parse cif files, if you try to parse a pdb with mmcif2dict a TypeError will be raised

structure_type :: string, values: biopython (default), prody, mmcif2dict

### 3.9.2 HPC_Drug/structures/protein.py

**Class Protein(HPC_Drug.structures.structure.Structure)**

This is the Protein class, subclasses HPC_Drug.structures.structure.Structure

**Method ___init___(self,**
**protein_id = None,**
**pdb_file = None,**
**structure = None,**

**substitutions_dict = None,**
**sulf_bonds = None,**
**seqres = None,**
**file_type = 'cif',**
**model = 0,**
**chain = 'A',**
**cys_dict = None,**
**gro_file = None,**
**top_file = None,**
**tpg_file = None,**
**prm_file = None)**
The constructor raises ValueError if no protein_id is given (string), if pdb_file == None pdb_file = self.protein_id.file_type, model must be an integer, chain upper case.
It will initialize self._ligands = [].

**Method add_ligand(self)** Adds a Ligand instance to self._ligands

**Method clear_ligands(self)** Clears ALL the ligands stored in self._ligands

**Method update_ligands(self, ligands)** Takes an iterable (list, tuple, etc...) containing HPC_Drug.structures.ligand.Ligand instances and ovewrites self._ligand with this new ones (any information about the old ones will be lost)

ligands :: iterable containing the new HPC_Drug.structures.ligand.Ligand instances

**Method get_ligand_list(self)** returns the list of ligands (self._ligands) already stored; it is a pointer to it, not a copy, so pay attention

### 3.9.3 HPC_Drug/structures/ligand.py

**Class Ligand(HPC_Drug.structures.structure.Structure)**

This is the Protein class, subclasses HPC_Drug.structures.structure.Structure

**Method ___init___(self,**
**resname = None,**
**file_type = 'pdb',**
**pdb_file = None,**

**structure = None,**
**resnum = None,**
**itp_file = None,**
**gro_file = None,**
**top_file = None,**
**tpg_file = None,**
**prm_file = None)**
resnum must be an integer, resname upper case, if pdb_file == None self.pdb_file = self.resname_lgand.file_type

### 3.9.4 HPC_Drug/structures/get_ligands.py

**Function get_ligands(Protein, ligand_resnames_resnums)**

Takes a HPC_Drug.structures.protein.Protein instance and a ligand_resnames_resnums and updates Protein._ligands with the newly created HPC_Drug.structures.ligand.Ligand instances

Protein :: HPC_Drug.structures.protein.Protein, Protein.file_type must be pdb !!!

ligand_resnames_resnums :: nested list of type [ ['ligand_resname', ligand_resnum], [...], ... ] if == None or == [] no HPC_Drug.structures.ligand.Ligand instance will be added to Protein._ligands

return Protein

## 3.10 HPC_Drug/auxiliary_functions

This folder contains some general use functions like get an absolute path from a relative one, get an iterable of a non iterable variable etc.

### 3.10.1 HPC_Drug/auxiliary_functions/path.py

**Function absolute_filepath(path)**

Takes a string and returns the absolute path of the file If the file does not exist raises a FileNotFoundError

path :: string

return absolute_path

**Function which(program)**

Uses shutil.which to get the absolute path of an executable that is in path example: path = which("python"), path = /usr/bin/python If the executable doesn't exist raises a OSError

program :: string

If you don't know if the program is in $PATH or not use absolute_programpath(program)

**Function absolute_programpath(program)**

It returns the absolute path to a program both if it is in $PATH or not

if the executable doesn't exist raises an OSError

program :: string

### 3.10.2 HPC_Drug/auxiliary_functions/get_iterable.py

**Function get_iterable(x)**

Returns an iterable, even if given a single value.

If x is a string returns (string,) even though a string is an iterable

### 3.10.3 HPC_Drug/auxiliary_functions/run.py

This file contains the functions needed to run external programs.

**Function subprocess_run(commands, shell = False, universal_newlines = False, error_string = "error during the call of an external program"cwd = os.getcwd())**

runs an external program using subprocess.run

if it fails will print the standard output, standard error and raise RuntimeError

commands :: list , it is the list of strings containing the command that subprocess.run will run

shell :: bool, dafault False, if == True the commands will be executed in the shell

universa_newlines :: bool, dafault False

error_string :: the sting to give to the RuntimeError as argument

cwd :: string , default the current working directory, it is the working directory for the child process

# Chapter 4

# Dependencies

Being a middleware this program has some important dependencies (this is not the list of pip requirements that will be found in the HPC_Drug repository):

- Biopython [9]
- numpy [11]
- OpenMM [7]
- pdbfixer [6]
- ProDy [10]
- scipy [12]
- ORAC [4]
- primadorac [3]
- Gromacs [5]
- Plumed [2]

# Bibliography


[1] Maurice Karrenbrock. HPC_Drug: a python application for Drug Development on High Performance Computing systems. Master's thesis, Università degli Studi di Firenze, Firenze, Italy, June 2020.

[2] Riccardo Capelli, Paolo Carloni, et al. Promoting transparency and reproducibility in enhanced molecular simulations. *Nature methods*, 16(FZJ-2019-05101):670–673, 2019.

[3] Piero Procacci. Primadorac: A free web interface for the assignment of partial charges, chemical topology, and bonded parameters in organic or drug molecules. *Journal of Chemical Information and Modeling*, 57(6):1240–1245, 2017. PMID: 28586207.

[4] Piero Procacci. Hybrid mpi/openmp implementation of the orac molecular dynamics program for generalized ensemble and fast switching alchemical simulations. *Journal of Chemical Information and Modeling*, 56(6):1117–1121, 2016. PMID: 27231982.

[5] Mark James Abraham, Teemu Murtola, Roland Schulz, Szilárd Páll, Jeremy C. Smith, Berk Hess, and Erik Lindahl. Gromacs: High performance molecular simulations through multi-level parallelism from laptops to supercomputers. *SoftwareX*, 1-2:19 – 25, 2015.

[6] Peter Eastman, Mark S. Friedrichs, John D. Chodera, Randall J. Radmer, Christopher M. Bruns, Joy P. Ku, Kyle A. Beauchamp, Thomas J. Lane, Lee-Ping Wang, Diwakar Shukla, Tony Tye, Mike Houston, Timo Stich, Christoph Klein, Michael R. Shirts, and Vijay S. Pande. Openmm 4: A reusable, extensible, hardware independent library for high performance molecular simulation. *Journal of Chemical Theory and Computation*, 9(1):461–469, 2013. PMID: 23316124.

[7] Peter Eastman, Jason Swails, John D Chodera, Robert T McGibbon, Yutong Zhao, Kyle A Beauchamp, Lee-Ping Wang, Andrew C Simmon-

ett, Matthew P Harrigan, Chaya D Stern, et al. Openmm 7: Rapid development of high performance algorithms for molecular dynamics. *PLoS computational biology*, 13(7):e1005659, 2017.

[8] Marina Macchiagodena, Marco Pagliai, Claudia Andreini, Antonio Rosato, and Piero Procacci. Upgrading and validation of the amber force field for histidine and cysteine zinc(ii)-binding residues in sites with four protein ligands. *Journal of Chemical Information and Modeling*, 59(9):3803–3816, 2019. PMID: 31385702.

[9] Peter J. A. Cock, Tiago Antao, Jeffrey T. Chang, Brad A. Chapman, Cymon J. Cox, Andrew Dalke, Iddo Friedberg, Thomas Hamelryck, Frank Kauff, Bartek Wilczynski, and Michiel J. L. de Hoon. Biopython: freely available Python tools for computational molecular biology and bioinformatics. *Bioinformatics*, 25(11):1422–1423, 03 2009.

[10] Ahmet Bakan, Lidio M. Meireles, and Ivet Bahar. ProDy: Protein Dynamics Inferred from Theory and Experiments. *Bioinformatics*, 27(11):1575–1577, 04 2011.

[11] Stéfan van der Walt, S. Chris Colbert, and Gaël Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30, 2011.

[12] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, CJ Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake Vand erPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1. 0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 2020.