Protons for OpenMM Documentation

Release 0.0.1

Bas Rustenburg, Greg Ross, John Chodera et al

Apr 10, 2018

Contents

Caution: This is module is undergoing heavy development. None of the API calls are final. This software is provided without any guarantees of correctness, you will likely encounter bugs.

If you are interested in this code, please wait for the official release to use it. In the mean time, to stay informed of development progress you are encouraged to:

- Follow this feed for updates on releases.
- Check out the github repository .

CHAPTER 1

Table of contents

1.1 Introduction

1.1.1 What is protons?

Protons is a python module that implements a constant-pH molecular dynamics scheme for sampling protonation states and tautomers of amino acids and small molecules in OpenMM.

The codebase is currently under development by the Chodera lab at Memorial Sloan Kettering Cancer center. There is no official release of the code yet. If you are interested in the development of this code, you can watch the repository on Github.

1.1.2 Installing Protons

Follow these instructions to install `protons`, and its dependencies.

Install using conda

The recommended way to install `protons` is through conda. You can obtain conda by installing the Anaconda python distribution. For instructions on how to use conda, please see this page to get you started.

As part of the first release, we will start building conda packages for `protons`. You will be able to install `protons` using the following conda command.

```
conda install -c omnia protons
```

This will install the latest version of `protons` from the Omnia channel.

Note: Currently, no official release conda packages are being built. A development release can be installed using the `-c omnia/label/dev` channel instead. Note that the `dev` releases may include insufficiently tested features.

Install latest development version

Note: This is only recommended for those developing the codebase.

You can install the latest development version using the conda recipe present in the Github _repository. First, install conda, and acquire conda-build, which is necessary to compile a conda package.

Download a copy of the repository, and cd into it.

```
git clone https://github.com/choderalab/protons.git
cd protons
```

If you are using bash, you can then use the following commands

```
 # Create an environment for development
- conda create --yes -n protons-development python=3.6
- source activate protons-development
- conda config --add channels omnia
# Add dev channels
- conda config --add channels omnia/label/dev
- conda build devtools/conda-recipe
- conda install --yes --use-local protons-dev
```

1.1.3 Testing your installation

Our library comes with an extensive test suite, designed to detect potential problems with your installation. After you've installed `protons`, it is recommended to verify that the code is working as expected.

To test the installation, run

```
py.test --pyargs protons
```

This requires that you have installed py.test in your python environment. The output of this command will tell you if any parts of the library are not working correctly. Note that it may take some time to complete the tests.

If a test fails, please try and verify whether your installation was successful. You may want to try reinstalling the library in a clean python environment and then testing it.

1.2 Preparing systems for constant-pH simulation

Several steps are necessary to prepare a simulation system for simulation using `protons`. Most of `protons` relies on the OpenMM API for Amber files. In order to run constant-pH simulations, the `protons` package requires input files generated by several programs from the Ambertools suite.

These instructions assume that you have a `.pdb` file of your system that has all necessary components to start an MD simulation. If you need help adding missing residues, atoms, or other features to your system, consider reading the OpenMM Modeller documentation, or the Amber manual.

Using the examples shown below, we will demonstrate how to generate input for an implicit solvent constant-pH simulation, including

- An amber topology file (`.prmtop`), which contains the parameters and topology of the system.
- A coordinate file (`.inpcrd`), which contains the 3-dimensional coordinates of your system.

- A protonation state file (`.cpin`), which enumerate protonation states of amino acids.

1.2.1 Obtaining and using Ambertools

Ambertools provides a series of command-line utilities that aid in setting up simulation systems. We will be using these for the manipulation of `.pdb` and `.mol2` files, and for generating parameters for small molecules using the general Amber force field (GAFF). The instructions below have been tested with Ambertools16.

Ambertools16 can be obtained from the Amber website.

For any instructions relating to Ambertools programs, we refer to the Amber manual. This includes instructions on how to install Ambertools.

After following the installation procedure, please make sure you can access the following programs on your command-line:

- `tleap`,
- `cpinutil.py`,

and if you're planning to use small molecules:

- `antechamber`,
- `parmchk`.

We will be using these to prepare systems for constant-pH simulation in OpenMM.

1.2.2 Fixing residue names and atoms

Several modifications will need to be made to your input file, in order to rename residues, and prevent issues with preexisting hydrogen atoms.

Renaming residues

One of the first steps of preparing an input file for constant-pH simulation is to rename all residues in the pdb file to their most protonated form.

The shortlist for PDB residues that our code supports is as follows:

- Histidine (HIP)
- Aspartic acid (AS4)
- Glutamic acid (GL4)
- Cysteine (CYS)
- Lysine (LYS)
- Arginine (ARG)

Note: You may also need to rename any cysteine residues involved in disulfide bonds from `CYS` → `CYX`, for further processing steps.

Treating hydrogens

To properly add all necessary hydrogens to the system, it is easiest to first delete all current hydrogens in the system. Otherwise, improperly named hydrogens may cause issues when defining the topology of the system based on the `.pdb` file. Missing hydrogens will be added back in the next preparation step, so it is safe to delete them at this stage.

An example bash script for processing `.pdb` files

Calling the following shell script on your pdb file should help you prepare your `.pdb` file for simulation. It replaces the names of residues in-place in the file, and it removes hydrogens from your file.

```
pdbfile="input.pdb"

# Rename to constph residues
sed -i 's/HIE/HIP/g' ${pdbfile}
sed -i 's/HID/HIP/g' ${pdbfile}
sed -i 's/HIS/HIP/g' ${pdbfile}
sed -i 's/ASP/AS4/g' ${pdbfile}
sed -i 's/GLU/GL4/g' ${pdbfile}

# Remove hydrogens (print lines with atom names not starting with H)
awk '$3 !~ /^ *H/' ${pdbfile} > tmp && mv tmp ${pdbfile}
```

Tools like `sed` and `awk` can be very useful for manipulating text files quickly, though I recommend that you verify the output manually afterwards.

1.2.3 Writing out coordinate and parameter files using Leap

The next step in system preparation is generating the Amber coordinate (`.inpcrd`), and the topology and parameters files (`.prmtop`). This can be done using the command-line utility `tleap`, which is available as part of ambertools.

These instructions will suffice if your system does not include any small molecules. If your system contains a ligand, please see the *next section*.

You can use `tleap` interactively on the command line. Just type

```
tleap
```

And you will see output similar to this

```
-bash-4.1$ tleap
-I: Adding /home/user/bin/../dat/leap/prep to search path.
-I: Adding /home/user/bin/../dat/leap/lib to search path.
-I: Adding /home/user/bin/../dat/leap/parm to search path.
-I: Adding/home/user/bin/../dat/leap/cmd to search path.

Welcome to LEaP!
(no leaprc in search path)
>
```

You can start typing your commands line by line. Alternatively, you can store commands in a text file, and then use

```
tleap -f tleap.txt
```

and tleap will run the specified commands automatically. Tleap output can be rather verbose. It is recommended to write the output to file, so you can verify that all steps were executed correctly.

Here is a bash example:

```
tleap -f tleap.in >> tleap.out 2>&1
```

You can rename the `.out` file to anything of your choosing.

Tleap commands

The following sequence of commands should do for a simple pdb file containing one protein structure.

```
# Load constant ph parameters
source leaprc.constph

# Load the PDB file, rename it to your input file
protein = loadPDB input.pdb

# Validate the input
check protein

# Calculate the total charge, for logging purposes
charge protein

# Write parameters.
saveAmberParm protein complex.prmtop complex.inpcrd

# Write PDB files, optional
savepdb protein complex.pdb

# Exit, make sure not to forget this part
quit
```

If you need to perform other steps to prepare your system for simulation, please read the Amber manual.

Validating tleap results

If you run interactively, tleap should provide error messages on screen. The output can be rather verbose, so make sure that your terminal is configured to scroll back far.

Alternatively, if you run using an input file, make sure that `tleap` ran successfully.

I often write output to a log file, and check the log file for errors. Here is a short bash snippet that does the trick.

```
tleap -f tleap.in >> tleap.out 2>&1

# There might be other error clues. This method isn't fail safe.
tleap_result=$(grep "usage" tleap.out || grep -i "error" tleap.out)

# As long as the grep results are empty
 if [ -z "$tleap_result" ]
 then
   echo -e "\e[32mTleap looks successful. Still, act cautious. She's a slippery one.
↪\e[39m"
 else
   echo -e "\e[31mCaught an error in Tleap. Tough luck, buddy.\e[39m"
   echo $tleap_result
 fi
```

This procedure generates three different files:

- `complex.prmtop`, an Amber topology file which contains the topology and parameters of the protein system.
- `complex.inpcrd`, a file containing the coordinates of all atoms in the system
- `complex.pdb`, this file is optional. You can use a `pdb` file in software such as PyMOL, to verify that the prepared structure doesn't contain mistakes.

You will be needing these to run your OpenMM script.

1.2.4 Including ligands in your system

Warning:

- Ligand support is a work in progress. We've experienced system instability with small molecules in implicit solvent simulations.

If you have a ligand, you will have to prepare your ligand using `antechamber`, and `parmchk`. This is used to generate two files

- `ligand.gaff.mol2`, a mol2 file with GAFF atom types.
- `ligand.gaff.frcmod`, an frcmod file with GAFF parameters for the ligand.

Here is an example of how to run `antechamber` and `parmchk`.

```
antechamber -i ligand.mol2 -fi mol2 -o ligand.gaff.mol2 -fo mol2
parmchk -i ligand.gaff.mol2 -o ligand.gaff.frcmod  -f mol2
```

You may wish to explore the advanced options of `antechamber` if you need to generate charges for your ligands. If you want to generate charges in another program, using a `.mol2` file should allow you to maintain those charges. Now that you've generated parameters for your ligand, these files then need to be added to your leap setup.

Here is an example leap script.

```
# Load constant ph parameters
source leaprc.constph

# Gaff params
source leaprc.gaff

# Load ligand parameters
ligand = loadMol2 ligand.gaff.mol2
loadAmberParams ligand.gaff.frcmod

# Load the PDBs
protein = loadPDB protein.pdb

# Combine into one complex
complex = combine { protein ligand }

# Validate the input
check complex

# Calculate the total charge, for logging purposes
charge complex
```

(continues on next page)

(continued from previous page)

```
# Write parameters.
saveAmberParm  complex  complex.prmtop complex.inpcrd

# Write PDB files
savepdb protein complex.pdb

# Exit, make sure not to forget this part
quit
```

Todo:

- In the current version of the code, ligands can not be treated using constant-pH methodologies.

1.2.5 Generating parameters for amino acid protonation states

The last step to generate input for the constant-pH simulation is to generate a `.cpin` file for your protein. This file contains the parameters for the different protonation states of the amino acids in the system.

A `.cpin` file can be generated by `cpinutil.py`, which is also distributed as part of Ambertools.

```
cpinutil.py -resnames HIP GL4 AS4 TYR LYS CYS -p complex.prmtop -o complex.cpin
```

1.3 Running a constant-pH MD simulation

The `protons` package extends OpenMM with the capability to alter the configuration of protons in simulation. This allows for sampling over multiple protonation states and prototropic tautomers of amino acid residues and small molcules. These effects have been shown as important in multiple biological systems (*[Czodrowski2007a]*, *[Czodrowski2007b]*, *[Steuber2007]*, *[Neeb2014]*), and now you can include them in OpenMM simulations.

1.3.1 Setting up the `AmberProtonDrive` class

In order to run a simulation containing multiple protonation states and tautomers, you can use *`AmberProtonDrive`*. This object is responsible for keeping track of the current system state, and updates the OpenMM context with the correct parameters. It uses an instantaneous Monte Carlo sampling method *[Mongan2004]* to update implicit solvent systems. Explicit solvent systems can be updated using NCMC *[Stern2007]*, *[Nilmeier2011]*, *[Chen2015]*. The driver maintains a dictionary of all possible protonation states and tautomers of each residue in the simulation system, and their parameters in the AMBER99 constant-pH force field *[Mongan2004]*.

To instantiate the *`AmberProtonDrive`*, you need a prmtop file (loaded using `simtk.openmm.app.AmberPrmtopFile`), and an OpenMM system (`simtk.openmm.openmm.System`).

Additionally, a `.cpin` file (generated by `cpinutil.py`, which is part of Ambertools) is needed to provide information on amino acid parameters. Please see the *Preparing systems for constant-pH simulation* section for instructions how to generate input files.

In order to perform NCMC, you need to provide the integrator that you will use for the regular MD part as well. The implementation of NCMC uses a Velocity Verlet integrator, so make sure that your integrator is not timewise incompatible like for instance Leapfrog integrators.

```
from protons import AmberProtonDrive
from simtk import unit, openmm
from simtk.openmm import app
from openmmtools.integrators import VelocityVerletIntegrator

# Load a protein system
prmtop = app.AmberPrmtopFile('protein.prmtop')
cpin_filename = 'protein.cpin'

# System settings
pH = 7.4
temperature = 300.0 * unit.kelvin

# Define an integrator for the system
integrator = VelocityVerletIntegrator(1.0 * unit.femtoseconds)

# Create an implicit solvent system from the AMBER prmtop file
system = prmtop.createSystem(implicitSolvent=app.OBC2, nonbondedMethod=app.NoCutoff,␣
↪constraints=app.HBonds)
# Create the driver that will update the protons on the system and track its state
driver = AmberProtonDrive(system, temperature, pH, prmtop, cpin_filename, integrator,␣
↪pressure=None, ncmc_steps_per_trial=0, implicit=True)

# Create a simulation object using the correct integrator
simulation = app.Simulation(prmtop.topology, system, driver.compound_integrator,␣
↪platform)
```

Next, you will need to provide reference free energies for each residue in solvent.

1.3.2 Solvent reference state calibrations

In order to simulate a protein or small molecule with multiple protonation states and/or tautomers, it is necessary to calculate the free energy difference between a reference state. This free energy can depend on the temperature, solvent model, pressure, pH and other factors. It is therefore imperative that this is run whenever you've changed simulation settings.

Individual states are denoted with a subscript k. We use self-adjusted mixture sampling (SAMS) to calculate g_k, a reference free energy. The g_k s correct for (electrostatic) force field contribution to the free energy difference between the reference states, so that the populations produced in simulation match what is expected from the pH dependence, or tautomeric populations.

Residues

The package supports automatic g_k calculations the following residues by default, denoted by the residue name with the max number of protons added. The reference state is taken to be the state of a single capped amino acids in water.

- Glutamic acid, `GL4` (pKa=4.4)
- Aspartic acid, `AS4` (pKa=4.0)
- Histidine, `HIP` (pKa delta=6.5, pKa epsilon = 7.1)
- Tyrosine, `TYR` (pKa=9.6)
- Cysteine, `CYS` (pKa=8.5)
- Lysine, `LYS` (pKa=10.4)

To automatically calibrate all amino acids available in a system, one can use the `AmberProtonDrive.calibrate()` method.

The `AmberProtonDrive.calibrate()` method

The `AmberProtonDrive.calibrate()` method will set this up automatically for the settings you have provided.

```
calibration_results = driver.calibrate()
```

It will automatically perform a free energy calculation using self-adjusted mixture sampling (SAMS) to calculate the reference free energy for each state g_k. While this is conveniently carried out automatically, this may take quite some time (minutes to 2-hours on a GTX-Titan per unique residue type). We are experimenting with a setup that can perform calibration in parallel so that you can run calibration more efficiently. If you store these results, you can reload them in a subsequent run.

```
# Pre-calculated values
# temperature = 300.0 * unit.kelvin
# pressure = None
# timestep = 1.0 * unit.femtoseconds
# pH = 7.4
# Amber 99 constant ph residues

calibration_results = {'as4': np.array([3.98027947e-04,  -3.61785292e+01,  -3.
↪98046143e+01,
                                        -3.61467735e+01,  -3.97845096e+01]),
                       'cys': np.array([7.64357397e-02,   1.30386793e+02]),
                       'gl4': np.array([9.99500333e-04,  -5.88268681e+00,  -8.
↪98650420e+00,
                                        -5.87149375e+00,  -8.94086390e+00]),
                       'hip': np.array([2.39229276,   5.38886021,  13.12895206]),
                       'lys': np.array([9.99500333e-04,  -1.70930870e+01]),
                       'tyr': np.array([6.28975142e-03,   1.12467299e+02])}

driver.import_gk_values(calibration_results)
```

Warning: When reusing calibrated values, you must make sure that you are using the exact same force field, pH and other properties of the system. If you are not sure, we recommend that you rerun the calibration.

For more in depth explanation of the calibration procedure, please see advanced_calibration.

Now that g_k values have been calibrated, you are ready to run a simulation.

1.3.3 Running the simulation

After calibration, you can start running a simulation. Decide on the number of timesteps, and the frequency of updating the residue states. To propagate in regular dynamics, just use `simulation.step`. The residue states are updated using the `AmberProtonDrive.update()` method. This method selects new states using a Monte Carlo procedure, and modifies the parameters in your simulation context to reflect the selected states.

```
nupdates, mc_frequency = 10000, 6000

for iteration in range(1, nupdates):
```

(continues on next page)

(continued from previous page)

```
    simulation.step(mc_frequency) # MD
    driver.update(simulation.context)  # protonation
```

In this example, every 6000 steps of molecular dynamics, the residue states are driven once. This gets repeated for a total of 10000 iteration.

1.3.4 Tracking the simulation

This section and the API still need to be written.

1.3.5 Basic example

Below is a basic example of how to run a simulation using the AmberProtonDrive without using the calibration API.

```
from simtk import unit, openmm
from simtk.openmm import app
from protons import AmberProtonDrive
import numpy as np
from openmmtools.integrators import VelocityVerletIntegrator
from sys import stdout


# Import one of the standard systems.
temperature = 300.0 * unit.kelvin
timestep = 1.0 * unit.femtoseconds
pH = 7.4

platform = openmm.Platform.getPlatformByName('CUDA')

prmtop = app.AmberPrmtopFile('complex.prmtop')
inpcrd = app.AmberInpcrdFile('complex.inpcrd')
positions = inpcrd.getPositions()
topology = prmtop.topology
cpin_filename = 'complex.cpin'
integrator = VelocityVerletIntegrator(timestep)

# Create a system from the AMBER prmtop file
system = prmtop.createSystem(implicitSolvent=app.OBC2, nonbondedMethod=app.NoCutoff,
↪ constraints=app.HBonds)
# Create the driver that will track the state of the simulation and provides the␣
↪updating API
driver = AmberProtonDrive(system, temperature, pH, prmtop, cpin_filename,␣
↪integrator, pressure=None, ncmc_steps_per_trial=0, implicit=True)

# Create an OpenMM simulation object as one normally would.
simulation = app.Simulation(topology, system, driver.compound_integrator, platform)
simulation.context.setPositions(positions)
simulation.context.setVelocitiesToTemperature(temperature)

# pre-equilibrated values.
# temperature = 300.0 * unit.kelvin
# pressure = None
# timestep = 1.0 * unit.femtoseconds
# pH = 7.4
```

(continues on next page)

(continued from previous page)

```
  # Amber 99 constant ph residues, converged to threshold of 1.e-7

  calibration_results = {'as4': np.array([3.98027947e-04,  -3.61785292e+01,  -3.
↪98046143e+01,
                                          -3.61467735e+01,  -3.97845096e+01]),
                         'cys': np.array([7.64357397e-02,   1.30386793e+02]),
                         'gl4': np.array([9.99500333e-04,  -5.88268681e+00,  -8.
↪98650420e+00,
                                          -5.87149375e+00,  -8.94086390e+00]),
                         'hip': np.array([2.39229276,   5.38886021,  13.12895206]),
                         'lys': np.array([9.99500333e-04,  -1.70930870e+01]),
                         'tyr': np.array([6.28975142e-03,   1.12467299e+02])}

  driver.import_gk_values(calibration_results)

  # 60 ns, 10000 state updates
  niter, mc_frequency = 10000, 6000
  simulation.reporters.append(app.DCDReporter('trajectory.dcd', mc_frequency))

  for iteration in range(1, niter):
      simulation.step(mc_frequency) # MD
      driver.update(simulation.context)  # protonation
```

1.4 Examples

1.4.1 Running a calibration

Here is an example script that can be used to calculate the relative free energies of an amino acid in solvent. The most up to date version of this script can be found here.

First, we import the dependencies. One thing to note, when running protons simulations, load the protons `app` module, rather than `simtk.openmm.app`. This is an augmented version of the app layer, including some additional utilities for running constant-pH simulations.

```
import json
import os
import shutil
import signal
import sys

import netCDF4
import numpy as np
from openmmtools.integrators import (ExternalPerturbationLangevinIntegrator,
                                     LangevinIntegrator)
from protons import app
from protons.app import (ConstantPHCalibration, ForceFieldProtonDrive,
                         MetadataReporter, NCMCProtonDrive, NCMCReporter,
                         SAMSReporter, TitrationReporter)
from protons.app.logger import log, logging
from saltswap.wrappers import Salinator
from simtk import openmm as mm
from simtk import unit

log.setLevel(logging.DEBUG)
```

We define some useful functions and classes that help us deal with runtime limits of a compute cluster.

```
# Define what to do on timeout
class TimeOutError(RuntimeError):
    """This error is raised when an operation is taking longer than expected."""
    pass

def timeout_handler(signum, frame):
    """Handle a timeout."""
    log.warn("Script is running out of time. Attempting to exit cleanly.")
    raise TimeOutError("Running out of time, shutting down!")
```

These functions simplify serializing the simulation state at the end of the run, so you can resume where you left off.

```
def serialize_state(context, outputfile):
    """
    Serialize the simulation state to xml.
    """
    xmls = mm.XmlSerializer
    statexml = xmls.serialize(context.getState(getPositions=True, getVelocities=True))
    with open(outputfile,'w') as statefile:
        statefile.write(statexml)

def serialize_drive(drive, outputfile):
    """
    Serialize the drive residues to xml.
    """
    drivexml = drive.serialize_titration_groups()
    with open(outputfile, 'wb') as drivefile:
        drivefile.write(drivexml)

def serialize_sams_status(calibration, outputfile):
    """
    Serialize the state of the SAMS calibration as json
    """
    samsProperties = calibration.export_samsProperties()
    samsjson = json.dumps(samsProperties, sort_keys=True, indent=4, separators=(',',
↪': '))
    with open(outputfile, 'w') as samsfile:
        samsfile.write(samsjson)
```

Next, we define an integrator class, based on one of the integrators defined in `openmmtools`. We recommend using a BAOAB integrator for NCMC, as we expect it to lead to smaller errors in the protocol work than a more conventional langevin integrator.

```
class ExternalGBAOABIntegrator(ExternalPerturbationLangevinIntegrator):
    """
    Implementation of the gBAOAB integrator which tracks external protocol work.

    Parameters
    ----------
        number_R: int, default: 1
            The number of sequential R steps.  For instance V R R O R R V has number_
↪R = 2
        temperature : simtk.unit.Quantity compatible with kelvin, default: 298*unit.
↪kelvin
        The temperature.
        collision_rate : simtk.unit.Quantity compatible with 1/picoseconds, default:␣
↪1.0/unit.picoseconds
```

(continues on next page)

(continued from previous page)

```
        The collision rate.
        timestep : simtk.unit.Quantity compatible with femtoseconds, default: 1.
↪0*unit.femtoseconds
        The integration timestep.


    """

    def __init__(self, number_R_steps=1, temperature=298.0 * unit.kelvin,
                 collision_rate=1.0 / unit.picoseconds,
                 timestep=1.0 * unit.femtoseconds,
                 constraint_tolerance=1e-7
                 ):
        Rstep = " R" * number_R_steps

        super(ExternalGBAOABIntegrator, self).__init__(splitting="V{0} O{0} V".
↪format(Rstep),
                                                       temperature=temperature,
                                                       collision_rate=collision_rate,
                                                       timestep=timestep,
                                                       constraint_tolerance=constraint_
↪tolerance,
                                                       measure_shadow_work=False,
                                                       measure_heat=False,
                                                       )
```

Then, the last (and largest step), we define a main function that can read in a json file with simulation settings, sets up, and runs the simulation.

```
def main(jsonfile):
    """Main simulation loop."""

    settings = json.load(open(jsonfile))
    # Parameters include format strings that are used to format the names of input␣
↪and output files
    prms = settings["parameters"]

    # Input files
    inp = settings["input"]
    idir = inp["dir"].format(**prms)
    input_pdbx_file = os.path.join(idir, inp["calibration_system"].format(**prms))

    # If supplied, tell the code to find and load supplied ffxml file
    custom_xml_provided = False
    if "ffxml" in inp:
        custom_xml_provided = True


    # Load the PDBxfile and the forcefield files
    if custom_xml_provided:
        custom_xml = os.path.join(idir, inp["ffxml"].format(**prms))
        custom_xml = os.path.abspath(custom_xml)
        forcefield = app.ForceField('amber10-constph.xml', 'gaff.xml', custom_xml,
↪'tip3p.xml', 'ions_tip3p.xml')
    else:
        forcefield = app.ForceField('amber10-constph.xml', 'gaff.xml', 'tip3p.xml',
↪'ions_tip3p.xml')
```

(continues on next page)

(continued from previous page)

```
    pdb_object = app.PDBxFile(input_pdbx_file)

    # Prepare the Simulation
    topology = pdb_object.topology
    positions = pdb_object.positions

    # Quick fix for histidines in topology
    # Openmm relabels them HIS, which leads to them not being detected as
    # titratable. Renaming them fixes this.
    for residue in topology.residues():
        if residue.name == 'HIS':
            residue.name = 'HIP'

    # Naming the output files
    out = settings["output"]
    odir = out["dir"].format(**prms)

    if not os.path.isdir(odir):
        os.makedirs(odir)
    lastdir = os.getcwd()
    os.chdir(odir)

    name_netcdf = out["netcdf"].format(**prms)
    dcd_output_name = out["dcd"].format(**prms)

    # Files for resuming simulation
    resumes = out["resume_files"]

    output_context_xml = resumes["state"].format(**prms)
    output_drive_xml = resumes["drive"].format(**prms)
    output_calibration_json = resumes["calibration"].format(**prms)

    # Integrator options
    integrator_opts = prms["integrator"]
    timestep = integrator_opts["timestep_fs"] * unit.femtosecond
    constraint_tolerance = integrator_opts["constraint_tolerance"]
    collision_rate = integrator_opts["collision_rate_per_ps"] / unit.picosecond
    number_R_steps = 1

    # Steps of MD before starting the main loop
    num_thermalization_steps = int(prms["num_thermalization_steps"])
    # Steps of MD in between MC moves
    steps_between_updates = int(prms["steps_between_updates"])

    ncmc = prms["ncmc"]
    counterion_method = ncmc["counterion_method"].lower()
    if counterion_method not in ["chen-roux", "chenroux", "background"]:
        raise ValueError("Invalid ncmc counterion method, {}. Please pick Chen-Roux␣
↪or background.".format(counterion_method))
    ncmc_steps_per_trial = int(ncmc["steps_per_trial"])
    prop_steps_per_trial = int(ncmc["propagations_per_step"])
    total_iterations = int(prms["total_attempts"])

    # settings for minimization
    minimization = prms["minimization"]
    pre_run_minimization_tolerance = float(minimization["tolerance_kjmol"]) * unit.
↪kilojoule / unit.mole
```

(continues on next page)

(continued from previous page)

```
    minimization_max_iterations = int(minimization["max_iterations"])

    # SAMS settings
    sams = prms["SAMS"]
    beta_burnin = sams["beta_sams"]
    flatness_criterion = sams["flatness_criterion"]

    # Script specific settings

    # Register the timeout handling
    signal.signal(signal.SIGALRM, timeout_handler)

    script_timeout = 428400 # 119 hours

    # Platform Options

    platform = mm.Platform.getPlatformByName('CUDA')
    properties = {'CudaPrecision': 'mixed', 'DeterministicForces': 'true',
↪'CudaDeviceIndex': os.environ['CUDA_VISIBLE_DEVICES']}
    # System Configuration
    sysprops = prms["system"]
    nonbondedMethod = app.PME
    constraints = app.HBonds
    rigidWater = True
    ewaldErrorTolerance = float(sysprops["ewald_error_tolerance"])
    barostatInterval =  int(sysprops["barostat_interval"])
    switching_distance = float(sysprops["switching_distance_nm"]) * unit.nanometers
    nonbondedCutoff = float(sysprops["nonbonded_cutoff_nm"]) * unit.nanometers
    pressure = float(sysprops["pressure_atm"]) * unit.atmosphere
    temperature = float(sysprops["temperature_k"]) * unit.kelvin

    system = forcefield.createSystem(topology, nonbondedMethod=nonbondedMethod,␣
↪constraints=constraints,
                                     rigidWater=rigidWater,␣
↪ewaldErrorTolerance=ewaldErrorTolerance, nonbondedCutoff=nonbondedCutoff)

    #
    for force in system.getForces():
        if isinstance(force, mm.NonbondedForce):
            force.setUseSwitchingFunction(True)

            force.setSwitchingDistance(switching_distance)

    # NPT simulation
    system.addForce(
        mm.MonteCarloBarostat(
            pressure,
            temperature,
            barostatInterval))


    integrator = ExternalGBAOABIntegrator(number_R_steps=number_R_steps,␣
↪temperature=temperature, collision_rate=collision_rate, timestep=timestep,␣
↪constraint_tolerance=constraint_tolerance)
    ncmc_propagation_integrator = ExternalGBAOABIntegrator(number_R_steps=number_R_
↪steps, temperature=temperature, collision_rate=collision_rate, timestep=timestep,␣
↪constraint_tolerance=constraint_tolerance)
```

(continues on next page)

(continued from previous page)

```
    # Define a compound integrator
    compound_integrator = mm.CompoundIntegrator()
    compound_integrator.addIntegrator(integrator)
    compound_integrator.addIntegrator(ncmc_propagation_integrator)
    compound_integrator.setCurrentIntegrator(0)

    if custom_xml_provided:
        driver = ForceFieldProtonDrive(temperature, topology, system, forcefield, [
↪'amber10-constph.xml', custom_xml], pressure=pressure,
                                       perturbations_per_trial=ncmc_steps_per_trial,␣
↪propagations_per_step=prop_steps_per_trial)
    else:
        driver = ForceFieldProtonDrive(temperature, topology, system, forcefield, [
↪'amber10-constph.xml'], pressure=pressure,
                                       perturbations_per_trial=ncmc_steps_per_trial,␣
↪propagations_per_step=prop_steps_per_trial)

    # Assumes calibration residue is always the last titration group
    calibration_titration_group_index = len(driver.titrationGroups) - 1

    # Define residue pools
    pools = {'calibration' : [calibration_titration_group_index]}
    driver.define_pools(pools)
    # Create SAMS sampler
    simulation = app.ConstantPHCalibration(topology, system, compound_integrator,␣
↪driver, group_index=calibration_titration_group_index, platform=platform,␣
↪platformProperties=properties, samsProperties=sams)
    simulation.context.setPositions(positions)

    # After the simulation system has been defined, we can add salt to the system␣
↪using saltswap.
    salinator = Salinator(context=simulation.context,
                          system=system,
                          topology=topology,
                          ncmc_integrator=compound_integrator.getIntegrator(1),
                          salt_concentration=0.150 * unit.molar,
                          pressure=pressure,
                          temperature=temperature)
    salinator.neutralize()
    salinator.initialize_concentration()
    swapper = salinator.swapper

    # Protons can use the scheme from [Chen2015]_ to maintain charge neutrality.
    # If the Chen-Roux scheme is requested, attach swapper. Else, use neutralizing␣
↪background charge (happens under the hood of openmm).
    if counterion_method in ["chenroux", "chen-roux"]:
        simulation.drive.attach_swapper(swapper)

    # Minimize the initial configuration to remove bad contacts
    simulation.minimizeEnergy(tolerance=pre_run_minimization_tolerance,␣
↪maxIterations=minimization_max_iterations)
    # Slightly equilibrate the system, detect early issues.
    simulation.step(num_thermalization_steps)

    # Add reporters, these write out simulation data at regular intervals
    dcdreporter = app.DCDReporter(dcd_output_name, int(steps_between_updates/10))
```

(continues on next page)

(continued from previous page)

```
    ncfile = netCDF4.Dataset(name_netcdf, "w")
    metdatarep = MetadataReporter(ncfile)
    ncmcrep = NCMCReporter(ncfile,1)
    titrep = TitrationReporter(ncfile,1)
    simulation.reporters.append(dcdreporter)
    simulation.update_reporters.append(metdatarep)
    simulation.update_reporters.append(ncmcrep)
    simulation.update_reporters.append(titrep)
    samsrep = SAMSReporter(ncfile,1)
    simulation.calibration_reporters.append(samsrep)

    # MAIN SIMULATION LOOP STARTS HERE

    # Raises an exception if the simulation runs out of time, so that the script can␣
↪be killed cleanly from within python
    signal.alarm(script_timeout)

    try:
        for i in range(total_iterations):
            log.info("Iteration %i", i)
            if i == 5:
                log.info("Simulation seems to be working. Suppressing debugging info.
↪")
                log.setLevel(logging.INFO)
            # Regular MD
            simulation.step(steps_between_updates)
            # Update protonation state
            simulation.update(1, pool='calibration')
            # Adapt SAMS weight
            simulation.adapt()
        # Reset timer
        signal.alarm(0)

    except TimeOutError:
        log.warn("Simulation ran out of time, saving current results.")

    finally:
        # export the context
        serialize_state(simulation.context,output_context_xml)
        # export the driver
        serialize_drive(simulation.drive, output_drive_xml)
        # export the calibration status
        serialize_sams_status(simulation, output_calibration_json)


        ncfile.close()
        os.chdir(lastdir)
```

After this large function has been defined, the last bit that is remained is setting up the (very minimal) command line interface.

```
if __name__ == "__main__":
    if len(sys.argv) != 2:
        print("Please provide a single json file as input.")
    else:
        # Provide the json file to main function
        main(sys.argv[1])
```

1.5 The ligutils submodule

This submodule will contain the features that enable parametrizing ligands for constant-ph simulation.

1.5.1 Treating small molecules

Todo:

- Small molecules are currently work in progress.
- Will need to add support to determine populations and molecular data from other sources than Epik.
- Adding openeye am1 bcc charge support as an optional feature for those that have openeye available

Several steps are needed in order to prepare a small molecule for simulation using the constant-pH code.

1. Protonation states and tautomers of the ligand need to be enumerated. This relies on Epik at the moment.
2. Molecule states need to be parametrized using antechamber.
3. All states need to be combined into a single residue template as an ffxml file.
4. Calibrate reference free energies of the molecule in solvent to return the expected state populations in the reference state.

This code provides an automated pipeline to sequentially perform all of these steps using this handy function!

1.6 API documentation

1.6.1 ProtonDrives

The main engine of the protons code is the `ProtonDrive`.

NCMCProtonDrive class

class protons.app.driver.**NCMCProtonDrive**(*temperature*, *topology*, *system*, *pressure=None*, *perturbations_per_trial=0*, *propagations_per_step=1*)

The NCMCProtonDrive is a base class Monte Carlo driver for protonation state changes and tautomerism in OpenMM.

Protonation state changes, and additionally, tautomers are treated using the constant-pH dynamics method of Mongan, Case and McCammon *[Mongan2004]*, or Stern *[Stern2007]* and NCMC methods from Nilmeier *[Nilmeier2011]*, and Chen and Roux *[Chen2015]*.

Members

__init__(*temperature*, *topology*, *system*, *pressure=None*, *perturbations_per_trial=0*, *propagations_per_step=1*)

Initialize a Monte Carlo titration driver for simulation of protonation states and tautomers.

Parameters

- **temperature** (*simtk.unit.Quantity compatible with kelvin*) – Temperature at which the system is to be simulated.

- **topology** (*protons.app.Topology*) – OpenMM object containing the topology of system
- **system** (*simtk.openmm.System*) – System to be titrated, containing all possible protonation sites.
- **pressure** (*simtk.unit.Quantity compatible with atmospheres, optional*) – For explicit solvent simulations, the pressure.
- **perturbations_per_trial** (*int, optional, default=0*) – Number of perturbation steps per NCMC switching trial, or 0 if instantaneous Monte Carlo is to be used.
- **propagations_per_step** (*int, optional, default=1*) – Number of propagation steps in between perturbation steps.

attach_context (*context*)

Attaches a context to the Drive. The Drive requires a context with an NCMC integrator to be attached before it is functional.

Parameters **context** (*simtk.openmm.Context*) – Context that has a compound integrator bound to it. The integrator with index 1 is used for NCMC. The NCMC integrator needs to be a CustomIntegrator with the following two properties defined: first_step: 0 or 1. 0 indicates the first step in an NCMC protocol and can be used for special actions required such as computing the energy prior to perturbation. protocol_work: double, the protocol work performed by external moves in between steps.

attach_swapper (*swapper: saltswap.swapper.Swapper, proposal: protons.app.proposals.SaltSwapProposal = None*)

Provide a saltswapper to enable maintaining charge neutrality.

Parameters

- **– a saltswap.Swapper object that is used for ion manipulation and bookkeeping.** (*swapper*) –
- **– optional, a SaltSwapProposal derived class that is used to select ions. If not provided it uses** (*proposal*) –
- **OneDirectionChargeProposal** (*the*) –

update (*proposal, residue_pool=None, nattempts=1*)

Perform a number of Monte Carlo update trials for the system protonation/tautomer states of multiple residues.

Parameters

- **proposal** (*_StateProposal derived class*) – Defines how to select residues for updating
- **residue_pool** (*str*) – The set of titration group incides to propose from. Groups can be defined using self.define_pools. If None, select from all titration groups uniformly.
- **nattempts** (*int, optional*) – Number of individual attempts per update.

Notes

The titration state actually present in the given context is not checked; it is assumed the ProtonDrive internal state is correct.

define_pools(*dict_of_pools*)
Specify named residue_pools/subgroups of residues that can be sampled from separately.

For instance, it might be useful to separate the protein from the ligand so you can sample the protonation state of one component of the system at a time.

Note that the indices are dependent on self.titrationGroups, not a residue index in the PDB or in OpenMM topology.

Parameters **dict_of_pools** (*dict of list of int*) – Provide a dictionary with named groups of residue indices.

Examples

residue_pools = dict{protein=list(range(34)),ligand=[34])

import_gk_values(*gk_dict*, *strict=False*)
Import precalibrated gk values. Only use this if your simulation settings are exactly the same.

If you changed any details, rerun calibrate instead!

Parameters

- **gk_dict** (*dict*) – dict of starting value g_k estimates in numpy arrays, with residue names as keys.
- **strict** (*bool, default False*) – If True, raises an error if gk values are specified for nonexistent residue.

adjust_to_ph(*pH*)
Apply the pH target weight correction for the specified pH.

For amino acids, the pKa values in app.pkas will be used to calculate the correction. For ligands, target populations at the specified pH need to be available, otherwise an exception will be thrown.

Parameters **– float, the pH to which target populations should be adjusted.** (*pH*) –

Raises ValueError - if the target weight for a given pH is not supplied.

serialize_titration_groups()
Store residues handled by the drive as xml.

Returns

Return type str - xml representation of the residues inside of the drive.

AmberProtonDrive class

class protons.app.driver.**AmberProtonDrive**(*temperature*, *topology*, *system*, *cpin_filename*, *pressure=None*, *perturbations_per_trial=0*, *propagations_per_step=1*)
The AmberProtonDrive is a Monte Carlo driver for protonation state changes and tautomerism in OpenMM. It relies on Ambertools to set up a simulation system, and requires a `.cpin` input file with protonation states.

Protonation state changes, and additionally, tautomers are treated using the constant-pH dynamics method of Mongan, Case and McCammon *[Mongan2004]*, or Stern *[Stern2007]* and NCMC methods from Nilmeier *[Nilmeier2011]*, and Chen and Roux *[Chen2015]*.

Members

__init__(*temperature*, *topology*, *system*, *cpin_filename*, *pressure=None*, *perturbations_per_trial=0*, *propagations_per_step=1*)

Initialize a Monte Carlo titration driver for simulation of protonation states and tautomers.

Parameters

- **temperature** (*simtk.unit.Quantity compatible with kelvin*) – Temperature at which the system is to be simulated.
- **topology** (*protons.app.Topology*) – OpenMM object containing the topology of system
- **system** (*simtk.openmm.System*) – System to be titrated, containing all possible protonation sites.
- **cpin_filename** (*string*) – AMBER 'cpin' file defining protonation charge states and energies of amino acids
- **pressure** (*simtk.unit.Quantity compatible with atmospheres, optional*) – For explicit solvent simulations, the pressure.
- **perturbations_per_trial** (*int, optional, default=0*) – Number of perturbation steps per NCMC switching trial, or 0 if instantaneous Monte Carlo is to be used.
- **propagations_per_step** (*int, optional, default=1*) – Number of propagation steps in between perturbation steps.
- **to do** (*Things*) –
- **----------** –
- **Generalize simultaneous_proposal_probability to allow probability of single, double, triple, etc. proposals to be specified?** (***) –

ForceFieldProtonDrive class

class protons.app.driver.**ForceFieldProtonDrive**(*temperature*, *topology*, *system*, *forcefield*, *ffxml_files*, *pressure=None*, *perturbations_per_trial=0*, *propagations_per_step=1*, *residues_by_name=None*, *residues_by_index=None*)

The ForceFieldProtonDrive is a Monte Carlo driver for protonation state changes and tautomerism in OpenMM. It relies on ffxml files to set up a simulation system.

Protonation state changes, and additionally, tautomers are treated using the constant-pH dynamics method of Mongan, Case and McCammon *[Mongan2004]*, or Stern *[Stern2007]* and NCMC methods from Nilmeier *[Nilmeier2011]* and Chen and Roux *[Chen2015]*.

Members

__init__(*temperature*, *topology*, *system*, *forcefield*, *ffxml_files*, *pressure=None*, *perturbations_per_trial=0*, *propagations_per_step=1*, *residues_by_name=None*, *residues_by_index=None*)

Initialize a Monte Carlo titration driver for simulation of protonation states and tautomers.

Parameters

- **temperature** (*simtk.unit.Quantity compatible with kelvin*) – Temperature at which the system is to be simulated.

- **topology** (*protons.app.Topology*) – Topology of the system
- **system** (*simtk.openmm.System*) – System to be titrated, containing all possible protonation sites.
- **ffxml_files** (*str or list of str*) – Single ffxml filename, or list of ffxml filenames containing protons information.
- **forcefield** (*simtk.openmm.app.ForceField*) – ForceField parameters used to make a system.
- **pressure** (*simtk.unit.Quantity compatible with atmospheres, optional*) – For explicit solvent simulations, the pressure.
- **perturbations_per_trial** (*int, optional, default=0*) – Number of steps per NCMC switching trial, or 0 if instantaneous Monte Carlo is to be used.
- **propagations_per_step** (*int, optional, default=1*) – Number of propagation steps in between perturbation steps.
- **residues_by_index** (*list of int*) – Residues in topology by index that should be treated as titratable
- **residues_by_name** (*list of str*) – Residues by name in topology that should be treated as titratable

Notes

If neither residues_by_index, or residues_by_name are specified, all possible residues with Protons parameters will be treated.

1.6.2 Simulation runners

To run a constant-pH simulation, some bookkeeping may need to be done. The simulation context needs to be linked to the residues and states bookkeeping, the frequency of updates needs to be set, data needs to be written out at regular intevals, et cetera.

To this end, two simulation runners are available in protons

ConstantPHSimulation

class protons.app.simulation.**ConstantPHSimulation**(*topology*, *system*, *compound_integrator*, *drive*, *move=None*, *pools=None*, *platform=None*, *platformProperties=None*, *state=None*)

ConstantPHSimulation is an API for running constant-pH simulation in OpenMM analogous to app.Simulation.

Members

__init__(*topology*, *system*, *compound_integrator*, *drive*, *move=None*, *pools=None*, *platform=None*, *platformProperties=None*, *state=None*)

Create a ConstantPHSimulation.

Parameters

- **topology** (*Topology*) – A Topology describing the the system to simulate

- **system** (*System or XML file name*) – The OpenMM System object to simulate (or the name of an XML file with a serialized System)
- **compound_integrator** (*openmm.CompoundIntegrator or XML file name*) – The OpenMM Integrator to use for simulating the System (or the name of an XML file with a serialized System) Needs to have 2 integrators. The first integrator is used for MD, the second integrator is used for NCMC. The NCMC integrator needs to be a CustomIntegrator with the following two properties defined:

  first_step: 0 or 1. 0 indicates the first step in an NCMC protocol and can be used for special actions required such as computing the energy prior to perturbation. protocol_work: double, the protocol work performed by external moves in between steps.
- **drive** (*protons ProtonDrive*) – A ProtonDrive object that can manipulate the protonation states of the system.
- **move** (*StateProposal, default None*) – An MC proposal move for updating titration states.
- **pools** (*dict, default is None*) – A dictionary of titration group indices to group together.
- **platform** (*Platform=None*) – If not None, the OpenMM Platform to use
- **platformProperties** (*map=None*) – If not None, a set of platform-specific properties to pass to the Context's constructor
- **state** (*XML file name=None*) – The name of an XML file containing a serialized State. If not None, the information stored in state will be transferred to the generated Simulation object.

step (*steps*)
Advance the simulation by integrating a specified number of time steps.

update (*updates*, *move=None*, *pool=None*)
Advance the simulation by propagating the protonation states a specified number of times.

Parameters

- **updates** (*int*) – The number of independent updates to perform.
- **move** (*StateProposal*) – Type of move to update system. Uses pre-specified move if not given.
- **pool** (*str*) – The identifier for the pool of residues to update.

ConstantPHCalibration

class protons.app.simulation.**ConstantPHCalibration**(*topology*, *system*, *compound_integrator*, *drive*, *group_index=0*, *move=None*, *pools=None*, *samsProperties=None*, *platform=None*, *platformProperties=None*, *state=None*)

ConstantPHCalibration is an API for calibrating a constant-pH free energy calculation that uses self-adjusted mixture sampling (SAMS) to calculate the relative free energy of different protonation states in a simulation system.

Members

__init__(*topology*, *system*, *compound_integrator*, *drive*, *group_index=0*, *move=None*, *pools=None*, *samsProperties=None*, *platform=None*, *platformProperties=None*, *state=None*)

Create a ConstantPHCalibration.

Parameters

- **topology** (*Topology*) – A Topology describing the the system to simulate
- **system** (*System or XML file name*) – The OpenMM System object to simulate (or the name of an XML file with a serialized System)
- **compound_integrator** (*openmm.CompoundIntegrator or XML file name*) – The OpenMM Integrator to use for simulating the System (or the name of an XML file with a serialized System) Needs to have 2 integrators. The first integrator is used for MD, the second integrator is used for NCMC. The NCMC integrator needs to be a CustomIntegrator with the following two properties defined:

  first_step: 0 or 1. 0 indicates the first step in an NCMC protocol and can be used for special actions required such as computing the energy prior to perturbation. protocol_work: double, the protocol work performed by external moves in between steps.
- **drive** (*protons ProtonDrive*) – A ProtonDrive object that can manipulate the protonation states of the system.
- **group_index** (*int, default 0*) – The index of the titratable group in Drive.titrationGroups.
- **move** (*StateProposal, default None*) – An MC proposal move for updating titration states.
- **pools** (*dict, default is None*) – A dictionary of titration group indices to group together.
- **samsProperties** (*dict, default is None*) – A dictionary with properties for the sams calibration. Used to set parameters for calibration or to resume.
- **platform** (*Platform=None*) – If not None, the OpenMM Platform to use
- **platformProperties** (*map=None*) – If not None, a set of platform-specific properties to pass to the Context's constructor
- **state** (*XML file name, default None*) – The name of an XML file containing a serialized State. If not None, the information stored in state will be transferred to the generated Simulation object.

step(*steps*)

Advance the simulation by integrating a specified number of time steps.

update(*updates*, *move=None*, *pool=None*)

Advance the simulation by propagating the protonation states a specified number of times.

Parameters

- **updates** (*int*) – The number of independent updates to perform.
- **move** (*StateProposal*) – Type of move to update system. Uses pre-specified move if not given.
- **pool** (*str*) – The identifier for the pool of residues to update.

adapt()

Adapt the weights for the residue that is being calibrated.

1.7 References

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Bibliography

[Czodrowski2007a] Czodrowski, Paul; Sotriffer, Christoph A.; Klebe, Gerhard (2007a): Atypical protonation states in the active site of HIV-1 protease: a computational study. In Journal of chemical information and modeling 47 (4), pp. 1590–1598. DOI: 10.1021/ci600522c.

[Czodrowski2007b] Czodrowski, Paul; Sotriffer, Christoph A.; Klebe, Gerhard (2007b): Protonation changes upon ligand binding to trypsin and thrombin: structural interpretation based on pK(a) calculations and ITC experiments. In Journal of molecular biology 367 (5), pp. 1347–1356. DOI: 10.1016/j.jmb.2007.01.022.

[Neeb2014] Neeb, Manuel; Czodrowski, Paul; Heine, Andreas; Barandun, Luzi Jakob; Hohn, Christoph; Diederich, Francois; Klebe, Gerhard (2014): Chasing protons: how isothermal titration calorimetry, mutagenesis, and pKa calculations trace the locus of charge in ligand binding to a tRNA-binding enzyme. In Journal of medicinal chemistry 57 (13), pp. 5554–5565. DOI: 10.1021/jm500401x.

[Steuber2007] Steuber, Holger; Czodrowski, Paul; Sotriffer, Christoph A.; Klebe, Gerhard (2007): Tracing changes in protonation: a prerequisite to factorize thermodynamic data of inhibitor binding to aldose reductase. In Journal of molecular biology 373 (5), pp. 1305–1320. DOI: 10.1016/j.jmb.2007.08.063.

[Mongan2004] Mongan J, Case DA, and McCammon JA. Constant pH molecular dynamics in generalized Born implicit solvent. J Comput Chem 25:2038, 2004. http://dx.doi.org/10.1002/jcc.20139

[Stern2007] Stern HA. Molecular simulation with variable protonation states at constant pH. JCP 126:164112, 2007. http://link.aip.org/link/doi/10.1063/1.2731781

[Nilmeier2011] Nonequilibrium candidate Monte Carlo is an efficient tool for equilibrium simulation. PNAS 108:E1009, 2011. http://dx.doi.org/10.1073/pnas.1106094108

[Greenwood2010] Greenwood, J. R.; Calkins, D.; Sullivan, A. P.; Shelley, J. C., "Towards the comprehensive, rapid, and accurate prediction of the favorable tautomeric states of drug-like molecules in aqueous solution," J. Comput. Aided Mol. Des., 2010, 24, 591-604

[Shelley2007] Shelley, J.C.; Cholleti, A.; Frye, L; Greenwood, J.R.; Timlin, M.R.; Uchimaya, M., "Epik: a software program for pKa prediction and protonation state generation for drug-like molecules," J. Comp.-Aided Mol. Design, 2007, 21, 681-691

[Epik2016] Schrödinger Release 2016-3: Epik, Schrödinger, LLC, New York, NY, 2016.

[Amber2016] D.A. Case, R.M. Betz, W. Botello-Smith, D.S. Cerutti, T.E. Cheatham, III, et al. (2016), AMBER 2016, University of California, San Francisco. http://ambermd.org/

[Wang2004] Wang, Junmei, et al. "Development and testing of a general amber force field." Journal of computational chemistry 25.9 (2004): 1157-1174.

[Chen2015] Chen, Yunjie, and Benoît Roux. "Constant-pH hybrid nonequilibrium molecular dynamics–monte carlo simulation method." Journal of chemical theory and computation 11.8 (2015): 3919-3931.

Index

Symbols

A

C

D

F

I

N

S

U