pAPRika Documentation

paprika

Aug 11, 2023

GETTING STARTED

pAPRika

An advanced toolkit for binding free energy calculations

pAPRika is a python toolkit for setting up, running, and analyzing free energy molecular dynamics simulations.

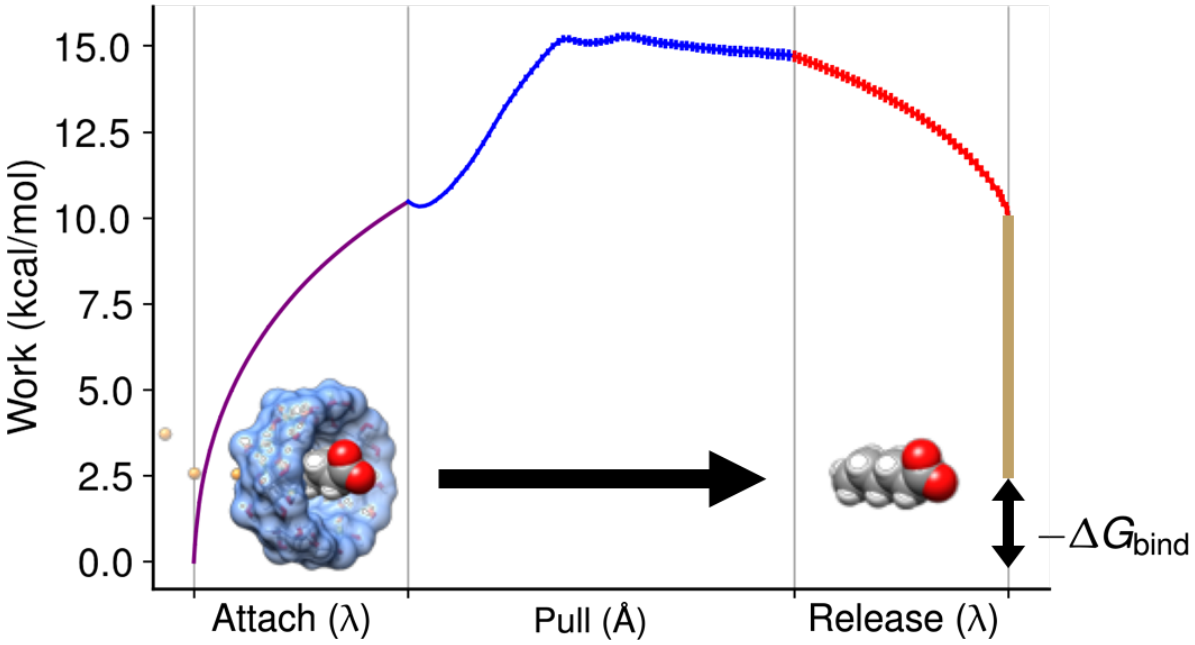
CHAPTER

ONE

BINDING FREE ENERGY

In the attach-pull-release (APR) method, a guest molecule is physically pulled out of the host molecule along a defined path by applying harmonic restraints. The binding free energy is then computed in terms of the sum of the work required for (1) *attaching* restraints to the bound complex, (2) *pulling* the guest molecule out of the host molecule and (3) *releasing* the restraints of the unbound complex:

$$\Delta G^{\circ}_{\text{bind}} = -(W_{\text{attach}} + W_{\text{pull}} + W_{\text{release-conf}} + W_{\text{release-std}}) \tag{1.1}$$



In the equation above, the work of releasing the restraints is split into two: $W_{\text{release-conf}}$ and $W_{\text{release-std}}$. These represents the work for releasing conformational restraints (applied to host or guest molecules) and the guest's translational and rotational restraints to the standard concentration, respectively. The translational and rotational degrees of freedom of the guest molecule is defined in spherical coordinates (r, θ, ϕ) and Euler angles (α, β, γ), respectively. Unlike the work for releasing conformational restraints, the work for releasing the guest's restraints to the standard concentration can be

evaluated analytically:

$$to$$

$$W_{\text{release-std}} = RT \ln\left(\frac{C^\circ}{8\pi^2}\right) + RT \ln\left(\int_0^\infty \int_0^\pi \int_0^{2\pi} e^{-\beta U(r,\theta,\phi)} r^2 \sin(\theta)\, d\phi\, d\theta\, dr\right) + RT \ln\left(\int_0^{2\pi} \int_0^\pi \int_0^{2\pi} e^{-\beta U(\alpha,\beta,\gamma)} \sin(\theta)\, d\alpha\, d\beta\, d\gamma\right) \quad (1.3)$$

=

$$+ RT \ln\left(\frac{C^\circ}{8\pi^2}\right) + RT \ln\left(\int_0^\infty \int_0^\pi \int_0^{2\pi} e^{-\beta U(r,\theta,\phi)} r^2 \sin(\theta)\, d\phi\, d\theta\, dr\right) \; RT \ln\left(\int_0^{2\pi} \int_0^\pi \int_0^{2\pi} e^{-\beta U(\alpha,\beta,\gamma)} \sin(\theta)\, d\alpha\, d\beta\, d\gamma\right) \quad (1.3)$$

For other terms in equation (1.1), the calculation is broken up into a series of independent windows. During the attach and release phase, the force constant of the restraints increases and decreases, respectively. In the pull phase, the distance between the guest and host is discretized from point A (bound) to point B (unbound) by changing the equilibrium position of the distance restraint. The work for each phase can be estimated using either *thermodynamic integration* (TI) or the *multistate-Bennett-Acceptance-Ratio* (MBAR).

$$to$$

$$W_{\text{attach,release}} = \int_0^1 \langle F \rangle_\lambda \, d\lambda \qquad W_{\text{pull}} = \int_A^B \langle F \rangle_r \, dr \quad (1.5)$$

=

$$= \int_0^1 \langle F \rangle_\lambda \, d\lambda W_{\text{pull}} \int_A^B \langle F \rangle_r \, dr \tag{1.5}$$

pAPRika estimates the standard error of the mean (SEM) with either *block data* analysis or *autocorrelation*.

CHAPTER

TWO

BINDING ENTHALPY

The binding enthalpy is the difference in the partial molar enthalpies of the bound complex and the separated molecules. Setting up binding enthalpy calculations is more straightforward than the binding free energy if we use the *direct* method. In the *direct* method, the binding enthalpy is obtained from the difference in the mean potential energy of the bound and unbound complex.

$$\Delta H_{\text{bind}} = \langle U_{\text{bound}} \rangle - \langle U_{\text{unbound}} \rangle \tag{2.1}$$

In the context of APR simulations, the mean potential energy is estimated from the first and last window. These windows, however, will need to be run for much longer until the fluctuation of the mean potential energy decreases below a desired value. Another way to estimate the binding enthalpy is with the *van't Hoff* method:

$$\ln K_{\text{eq}} = -\frac{\Delta H}{RT} + \frac{\Delta S}{R} \tag{2.2}$$

Here, the binding enthalpy is obtained by a linear regression of $\ln K_{\text{eq}}$ vs $1/T$. The equilibrium constant will need to be evaluated at different temperatures in order the estimate the binding enthalpy with the *van't Hoff* method.

CHAPTER

THREE

SUPPORTED MOLECULAR DYNAMICS ENGINES

Currently, we can use *pAPRika* to set up, perform, and analyze APR simulations with the AMBER, OpenMM, GROMACS, and NAMD programs. *pAPRika* supports these MD programs by providing a python-based wrapper for running the simulations except for OpenMM. These MD programs provide their own interface for restraints (e.g., AMBER uses NMR-style restraints). However, *pAPRika* also provides modules for generating APR restraints in Plumed and Colvars format, which can interface with various MD programs.

Future releases of *pAPRika* will include support for other MD engines like LAMMPS. Plumed is supported by all of the listed MD programs below, but for NAMD, only NVT simulations can be performed. For running NPT simulations in NAMD, however, we can use the Colvars module. Also, a pipeline for running double-decoupling calculations with *pAPRika* is currently in development and will be released soon.

Table 1: MD engines that are supported in *pAPRika* as simulation wrappers (or planned) and the respective restraint modules.

MD Engine	Restraints Module			
	NMR	XML	Plumed	Colvars
AMBER	✓	×	✓	×
OpenMM	×	✓	✓**	×
GROMACS	×	×	✓	×
NAMD	×	×	✓	✓
LAMMPS*	×	×	✓*	✓*

** Currently not supported but will be available in future releases. ** Supported but have not yet been tested.*

CHAPTER

FOUR

LICENSE

pAPRika is licensed under the BSD 3-Clause license.

CHAPTER

FIVE

REFERENCES

1. Velez-Vega, C. & Gilson, M. K. Overcoming Dissipation in the Calculation of Standard Binding Free Energies by Ligand Extraction. J. Comput. Chem. 34, 2360–2371 (2013).
2. Fenley, A. T., Henriksen, N. M., Muddana, H. S. & Gilson, M. K. Bridging calorimetry and simulation through precise calculations of cucurbituril-guest binding enthalpies. J. Chem. Theory Comput. 10, 4069–4078 (2014).
3. Henriksen, N. M., Fenley, A. T. & Gilson, M. K. Computational Calorimetry: High-Precision Calculation of HostGuest Binding Thermodynamics. J. Chem. Theory Comput. 11, 4377–4394 (2015).

5.1 Installation

We recommend installing *pAPRika* in a fresh `conda` environment if possible. There are three ways to install this package:

5.1.1 Installing from Conda

To install the latest release of *paprika* from conda, run:

```
conda install -c conda-forge paprika
```

In order to use all features of *paprika*, you must either have `AmberTools` (http://ambermd.org/AmberTools.php) in your *$PATH* or separately install `AmberTools` with the command below if it is not already installed in your environment:

```
conda install -c conda-forge ambertools=22
```

Optional dependencies

If you want to run simulations with Plumed-based restraints (needed for running APR in `GROMACS`) you can compile Plumed from source or install through conda:

```
conda install -c conda-forge plumed
```

Although `GROMACS` is available in conda (Bioconda), a version that is patched with Plumed is currently not available. Therefore, if you want to run `GROMACS` simulations in *paprika* you will need to compile from source manually (patched with Plumed).

5.1.2 Installing a stable version from source

To install the a stable version of *paprika* download the source code from Github. Then, unzip the files and change to the `paprika` directory:

```
tar -xvzf paprika-version.tgz
cd pAPRika-version/
```

Change the `name` field in `devtools/conda-envs/test_env.yaml` to be `paprika` and create the environment:

```
conda env create -f devtools/conda-envs/test_env.yaml
```

Activate the environment:

```
conda activate paprika
```

and install *paprika* in the environment:

```
pip install .
```

5.1.3 Installing latest from source

To install *paprika* with the latest features, clone the repository from the `master` branch on Github:

```
git clone https://github.com/GilsonLabUCSD/pAPRika.git
```

Change directory to the `paprika` folder, change the `name` field in `devtools/conda-envs/test_env.yaml` to `paprika` and create the conda environment:

```
conda env create -f devtools/conda-envs/test_env.yaml
```

Activate the environment:

```
conda activate paprika
```

and install *paprika* in the environment:

```
pip install .
```

5.2 Workflow

This page provides a brief explanation of the workflow to perform APR calculations with *pAPRika*. For more detail users are recommended to go through the tutorials, which details further on how to setup and run APR simulations from start to finish.

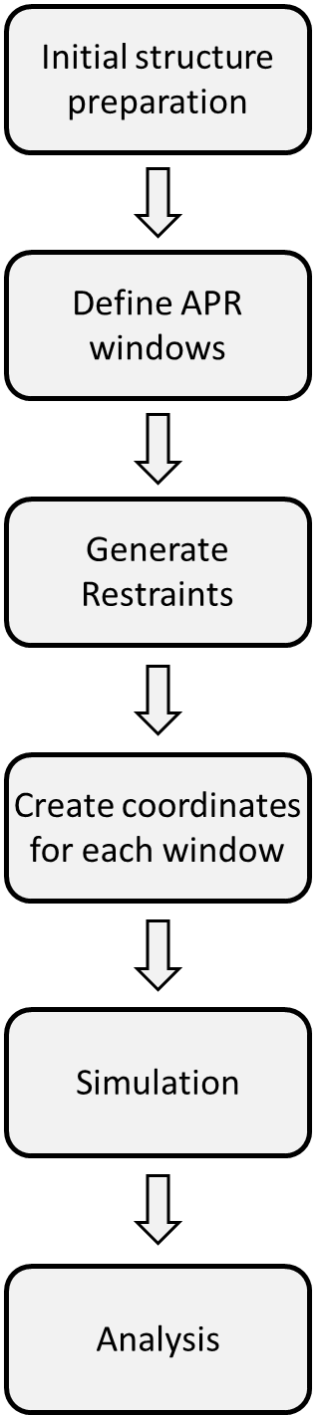


Fig. 1: Flowchart of the *paprika* workflow for a typical APR simulation.

5.2.1 Structure Preparation

Aligning the host-guest complex

The starting structure for the APR simulation can be configured with *paprika*. The APR calculation is most efficient in a rectangular box with the long axis parallel to the pulling axis (reduces the number of water molecules in the system). To make this easy to set up, the `Align` module provides functions to shift and orient a structure. For example, we can translate a structure to the origin and then orient the system to the z-axis by running

```
from paprika.build.align import *

translated_structure = translate_to_origin(structure)
aligned_structure = zalign(translated_structure, ":GST@C1", ":GST@C2")
```

Adding dummy atoms

To provide something to pull "against," we add dummy atoms that are fixed in place with strong positional restraints. These dummy atoms can be added to a the host-guest structure using the `Dummy Atoms` module in *paprika*.

```
from paprika.build.dummy import add_dummy

structure = dummy.add_dummy(structure, residue_name="DM1", z=-6.0)
structure = dummy.add_dummy(structure, residue_name="DM2", z=-9.0)
structure = dummy.add_dummy(structure, residue_name="DM3", z=-11.2, y=2.2)
structure.save("aligned_with_dummy.pdb", overwrite=True)
```

We will need the `mol2` and `frcmod` files for the dummy atoms, which we will need to generate the *AMBER* topology

```
dummy.write_dummy_frcmod(filepath="complex/dummy.frcmod")
dummy.write_dummy_mol2(residue_name="DM1", filepath="complex/dm1.mol2")
dummy.write_dummy_mol2(residue_name="DM2", filepath="complex/dm2.mol2")
dummy.write_dummy_mol2(residue_name="DM3", filepath="complex/dm3.mol2")
```

Building the topology

Finally, we can use the `tleap` wrapper to combine all of these components to generate the topology and coordinate files.

```
from paprika.build.system import TLeap

system = TLeap()
system.output_prefix = "host-guest-dum"
system.pbc_type = None
system.neutralize = False

system.template_lines = [
    "source leaprc.gaff",
    "HST = loadmol2 host.mol2",
    "GST = loadmol2 guest.mol2",
    "DM1 = loadmol2 dm1.mol2",
    "DM2 = loadmol2 dm2.mol2",
    "DM3 = loadmol2 dm3.mol2",
    "model = loadpdb aligned_with_dummy.pdb",
]
system.build()
```

Solvating the structure

The `TLeap` wrapper also provides an API for choosing water models when the user wants to solvate their structure. `tleap` provides a number of models for both 3-point and 4-point water models. There are also sub-types for each water model, e.g. for TIP3P we can choose to use the ForceBalance optimized variant called TIP3P-FB or the host-guest binding optimized model called Bind3P. To choose a water model for solvation we use the `set_water_model` method of the `TLeap` wrapper. The method requires the user to specify the water model and optionally the sub-type as the `model_type` attibute. The supported water models are:

- spc: None (SPCBOX), "flexible" (SPCFWBOX), "quantum" (QSPCFWBOX)
- opc: None (OPCBOX), "three-point" (OPC3BOX)
- tip3p: None (TIP3PBOX), "flexible" (TIP3PFBOX), "force-balance" (FB3BOX)
- tip4p: None (TIP4PBOX), "ewald" (TIP4PEWBOX), "force-balance" (FB4BOX)

Below is an example for solvating a system with 2000 TIP3P water molecules with `ForceBalance` optimized parameters.

```
from paprika.build.system import TLeap
from paprika.build.system.utils import PBCBox

system = TLeap()
system.output_prefix = "host-guest-dum"
system.pbc_type = PBCBox.rectangular
system.target_waters = 2000
system.set_water_model("tip3p", model_type="force-balance")

system.template_lines = [
    "source leaprc.gaff",
    "HST = loadmol2 host.mol2",
    "GST = loadmol2 guest.mol2",
    "DM1 = loadmol2 dm1.mol2",
    "DM2 = loadmol2 dm2.mol2",
    "DM3 = loadmol2 dm3.mol2",
    "model = loadpdb aligned_with_dummy.pdb",
]
system.build()
```

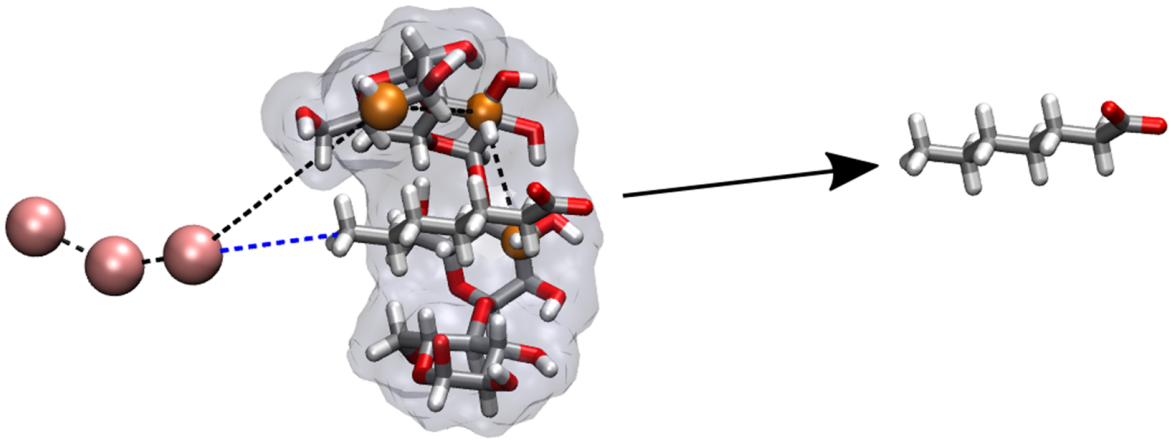
5.2.2 Defining Restraints

In APR calculations we apply restraints on the host (or protein) and the guest molecules. The restraints can be grouped into four categories: (1) *static restraints*, (2) *varying restraints*, (3) *wall restraints* and (4) *positional restraints*. The equilibrium target values and force constants can be specified as either a float or *Pint* quantity through the *openff-units* wrapper.

(1) Static Restraints

Static restraints do not change during the whole APR process and do not affect the free energy. We apply static restraints on the host (or protein) molecule to orient the host/protein degrees of freedom. The static restraints are composed of distance, angle, and torsional (DAT) restraints based on the choice of anchor atoms. For host-guest systems, we need to define three anchor atoms `[H1,H2,H3]` and combined with three dummy atoms `[D1,D2,D3]`, we apply a total of six static restraints on the host molecule (three for the translation and three for orientation).

To generate static restraints we use the function `static_DAT_restraints`. As an example, to apply a distance restraint on `D1` and `H1` with a force constant of 5 kcal/mol/^{2} we call



```
from openff.units import unit
from paprika.restraints import static_DAT_restraint

k_dist = 5.0 * unit.kcal / unit.mole / unit.angstrom ** 2

 dist_static = static_DAT_restraint(
     restraint_mask_list = [D1, H1],
     num_window_list = windows,  # list: [len(attach_lambda), len(pull_windows),␣
↪len(release_lambda)]
     ref_structure = structure,  # Structure file (PDB) or ParmEd structure object
     force_constant = k_dist,
 )
```

The equilibrium target for the harmonic restraint is estimated from the `ref_structure`.

(2) Varying Restraints

As the name suggests, these restraints change during the APR process. During the *attach* and *release* phases, the force constants of these restraints changes. In the *pull* phase, *varying restraints* can have their equilibrium position change, and this can be used as the restraint to pull the guest molecule out of the host molecule.

To generate *varying restraints*, we use the `DAT_restraint` class. The code below shows a restraints r that starts from 6.0 Å to 24 Å in the *pull* phase and stays restrained at 24 Å during the *release* phase.

```
from paprika.restraints import DAT_restraint

r_init = 6.0 * unit.angstrom
r_final = 24.0 * unit.angstrom
k_dist = 5.0 * unit.kcal / unit.mole / unit.angstrom ** 2

r = DAT_restraint()
r.mask1 = D1
r.mask2 = G1
r.topology = structure
r.auto_apr = True
r.continuous_apr = True
```

(continues on next page)

(continued from previous page)

```
r.attach["target"] = r_init
r.attach["fraction_list"] = attach_lambda
r.attach["fc_final"] = k_dist

r.pull["target_final"] = r_final
r.pull["num_windows"] = len(pull_windows)

r.release["target"] = r_final
r.release["fraction_list"] = [1.0] * len(release_lambda)
r.release["fc_final"] = k_dist

r.initialize()
```

Note: The `DAT_restraint` class can also be used to apply conformational restraints on the host and/or guest molecule. For example, distance "jack" and dihedral restraints can be applied to cucurbiturils and cyclodextrins host molecules, respectively, to make the binding site more accessible.

(3) Wall Restraints (optional)

Wall restraints are half-harmonic potentials that is useful for preventing guest molecules from leaving the binding site (for weak binding) or preventing the guest molecule from flipping during the attach phase. We still use the `DAT_restraint` class to generate the restraints but will use the `custom_restraint_values` method to generate the half-harmonic potential.

Note: `custom_restraint_values` follows the *AMBER* NMR-restraint format, see Chapter 27 in the AMBER20 manual for more details.

Below is an example for generating a *"lower wall"* restraint that prevents the angle of `[D2,G1,G2]` from decreasing below 91 degrees.

```
r_wall = 91.0 * unit.degrees
k_wall = 200.0 * unit.kcal / unit.mole / unit.radians ** 2

wall_orient = DAT_restraint()
wall_orient.mask1 = D1
wall_orient.mask2 = G1
wall_orient.mask3 = G2
wall_orient.topology = structure
wall_orient.auto_apr = True
wall_orient.continuous_apr = True

wall_orient.attach["num_windows"] = attach_fractions
wall_orient.attach["fc_initial"] = k_wall
wall_orient.attach["fc_final"] = k_wall

wall_orient.custom_restraint_values["r1"] = k_wall
wall_orient.custom_restraint_values["r2"] = 0.0
wall_orient.custom_restraint_values["rk2"] = k_wall
wall_orient.custom_restraint_values["rk3"] = 0.0
```

(continues on next page)

(continued from previous page)

```
wall_orient.initialize()
```

(4) Positional Restraints

Positional restraints in APR simulations are applied to the dummy atoms. Together with *static restraints*, this provides a laboratory frame of reference for the host-guest complex. Different MD programs handles *positional restraints* differently. For example, in `AMBER` you can define positional restraints in the input configuration file using the `ntr` keyword (Chapter 19 in the AMBER20 manual). For other programs like `GROMACS` and `NAMD` that uses `Plumed`, *positional restraints* can be applied using the method `add_dummy_atom_restraints()`.

Note: `tleap` may shift the coordinates of the system when it solvates the structure. Applying the *positional restraints* before the solvating the structure may lead to undesired errors during simulations. Therefore, special care needs to be taken when applying *positional restraints*. Take a look at tutorials *5* and *6* to see this distinction.

Creating the APR windows and saving restraints to file

To create the windows for the APR calculation we need to parse a *varying restraint* to the utility function `create_window_list`. This function will return a list of strings for the APR protocol

```
window_list = create_window_list(restraints_list)
window_list
["a000", "a001", ..., "p000", "p001", ...]
```

It may also be useful to save both the windows list and the restraints to a JSON file so you do not need to redefine again. The restraints can be saved to a JSON file using the utility function `save_restraints`.

```
from paprika.io import save_restraints
save_restraints(restraints_list, filepath="restraints.json")

import json
with open("windows.json", "w") as f:
    dumped = json.dumps(window_list)
    f.write(dumped)
```

Extending/adding more windows

Sometimes it may be necessary to add more windows in the APR calculation due to insufficient overlap between neighboring windows. For convenience we can add the windows at the end of the current list instead of inserting them in order. For example, let's say that we have a defined a restraint that spans from 8.4 to 9.8 Å and we want to add three windows between 8.6 and 9.0 Å.

```
r_restraint.pull
{'fc': 10.0,
 'target_initial': None,
 'target_final': None,
 'num_windows': None,
 'target_increment': None,
 'fraction_increment': None,
 'fraction_list': None,
 'target_list': array([8.4, 8.6, 9. , 9.4, 9.8])}
```

We will just need to append the *target_list* of this dictionary and reinitialize the restraints

```
r_restraint.pull["target_list"] = np.append(r_restraint.pull["target_list"], [8.7, 8.8,␣
↪8.9])
r_restraint.initialize()
r_restraint.pull
{'fc': 10.0,
 'target_initial': None,
 'target_final': None,
 'num_windows': None,
 'target_increment': None,
 'fraction_increment': None,
 'fraction_list': None,
 'target_list': array([8.4, 8.6, 9. , 9.4, 9.8, 8.7, 8.8, 8.9])}
```

We can save the updated restraints to a new file and pass it to the analysis script. The `fe_calc` class will take care of the window ordering thus there is no need to manually order the windows.

5.2.3 Running a Simulation

paprika provides wrappers with the `Simulate` module for a number of MD engines enabling us to run the simulations in python.

```
from paprika.simulate import AMBER

simulation = AMBER()
simulation.executable = "pmemd.cuda"
simulation.gpu_devices = "0"

simulation.path = "simulation"
simulation.prefix = "equilibration"
simulation.coordinates = "minimize.rst7"
simulation.ref = "host-guest-dum.rst7"
simulation.topology = "host-guest-dum.prmtop"
simulation.restraint_file = "disang.rest"

simulation.config_pbc_md()

# Positional restraints on dummy atoms
simulation.cntrl["ntr"] = 1
simulation.cntrl["restraint_wt"] = 50.0
simulation.cntrl["restraintmask"] = "'@DUM'"

print(f"Running equilibration in window {window}...")
simulation.run()
```

5.2.4 Analysis

Once the simulation is complete, the free energy can be obtained using the `Analysis` module, which will also estimate the uncertainties using the bootstrapping method. There are three types of methods that you can do with the `Analysis` module: (1) *thermodynamic integration* with *block-data* analysis ("ti-block"), (2) *multistate Benett-Acceptance-Ratio* with *block-data* analysis ("mbar-block"), and (3) *multistate Benett-Acceptance-Ratio* with *autocorrelation* analysis ("mbar-autoc").

```
from paprika.analysis import fe_calc
from paprika.io import load_restraints

restraints_list = load_restraints(filepath="restraints.json")

free_energy = fe_calc()
free_energy.prmtop = "host-guest-dum.prmtop"
free_energy.trajectory = 'production.nc'
free_energy.path = "windows"
free_energy.restraint_list = restraints_list
free_energy.collect_data()
free_energy.methods = ['ti-block']
free_energy.ti_matrix = "full"
free_energy.bootcycles = 1000
free_energy.compute_free_energy()
```

We can also estimate the free energy cost of releasing the restraints on the guest molecule semianalytically. To do this we need to extract the restraints that is specific to the guest molecule. The `extract_guest_restraints` function from the `restraints` module and pass this to the *analysis* object.

```
import parmed as pmd
from paprika.restraints.utils import extract_guest_restraints

structure = pmd.load_file("guest.prmtop", "guest.rst7", structure=True)
guest_restraints = extract_guest_restraints(structure, restraints_list, guest_resname=
↪"GST")
free_energy.compute_ref_state_work(guest_restraints)
```

The results are stored in the variable `results` as a python dictionary and you can save this to a JSON file.

```
print(free_energy.results["pull"]["ti-block"]["fe"])
-3.82139135698 kcal/mol

free_energy.save_results("APR_results.json")
```

The processed simulation data can also be saved to a JSON file so that you do not need to re-read the MD trajectories if you need to do further analysis.

```
free_energy.save_data("APR_simulation_data.json")
free_energy.collect_data_from_json("APR_simulation_data.json")
```

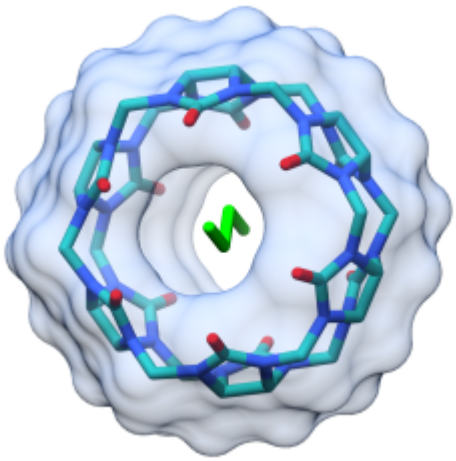
5.3 pAPRika tutorial 1 - An introduction to APR with Implicit solvent

In this example, we will setup and simulate butane (BUT) as a guest molecule for the host cucurbit[6]uril (CB6). CBs are rigid, symmetric, cyclic host molecules with oxygen atoms around the portal edge of the cavity. We will run the simulation in implicit solvent, using the Generalized-Born model, for speed and simplicity, using *AMBER*. This tutorial assumes familiarity with basic MD procedures.

5.3.1 Initial Setup

The very first step in the calculation is creating a coordinate file for the bound host-guest complex (we usually use PDB format for this part because it works well with `tleap`). This does not have to perfectly match the bound state by any means (we will later minimize and equilibrate), but this should be a reasonable illustration of the bound complex. This file can be created by hand (in a program like Chimera, VMD, or PyMOL) or by docking the guest into the host cavity (with MOE, AutoDock, DOCK, ...).

In this example, this file is called `cb6-but.pdb`, in the `complex/` directory, and this is what it looks like.



In addition to the coordinate file, we need separate `mol2` files for the host and guest molecule that contain the partial atomic charges. For cyclic hosts like CB6, you can specify either a single residue for the entire molecule (this is what we do here) or you can provide coordinates and charges for a single monomer and have `tleap` build the structure (this is a little more tricky).

The `mol2` files that we use here (`cb6.mol2` and `but.mol2`) were created by running `antechamber -fi pdb -fo mol2 -i <pdb> -o <mol2> -c bcc` (I added `pl 10` for the host, which reduces the number of paths that `antechamber` takes to traverse the host; see `antechamber` help for more information).

[4]:
```
import os
if not os.path.isdir("complex"):
    os.makedirs("complex")
data = "../../../paprika/data/cb6-but"
```

Create AMBER coordinate (`.rst7` or `.inpcrd`) and parameter files (`.prmtop` or `.topo`) for the host-guest complex

In this example, we will use GAFF parameters for both the host and guest. For the host, `parmchk2` has identified two parameters that are missing from GAFF and added the most similar ones into the supplementary `cb6.frcmod` file.

[5]:
```
from paprika.build.system import TLeap
```

[6]:
```
system = TLeap()
system.output_path = "complex"
system.pbc_type = None
system.neutralize = False

system.template_lines = [
    "source leaprc.gaff",
    f"loadamberparams {data}/cb6.frcmod",
    f"CB6 = loadmol2 {data}/cb6.mol2",
    f"BUT = loadmol2 {data}/but.mol2",
    f"model = loadpdb {data}/cb6-but.pdb",
    "check model",
    "savepdb model vac.pdb",
    "saveamberparm model vac.prmtop vac.rst7"
]
system.build()
```

After running `tleap`, it is always a good idea to check `leap.log`. `pAPRika` does some automated checking of the ouptut, but sometimes things slip through. (By default, `tleap` will append to `leap.log`.)

Prepare the complex for an APR calculation

Now we are ready to prepare the complex for the attach-pull-release calculation. This involves:

- Aligning the structure so the guest can be pulled along the z axis, and
- Adding dummy atoms that are used to orient the host and guest.

To access the host-guest structure in Python, we use the ParmEd `Structure` class. So we start by loading the vacuum model that we just created. Then, we need to define two atoms on the guest that are placed along the z axis. These should be heavy atoms on either end of the guest, the second atom leading the pulling.

These same atoms will be used later for the restraints, so I will name them `G1` and `G2`, using AMBER selection syntax.

Align the pulling axis

[7]:
```
import parmed as pmd
```

[8]:
```
structure = pmd.load_file("complex/vac.prmtop",
                          "complex/vac.rst7",
                          structure=True)
```

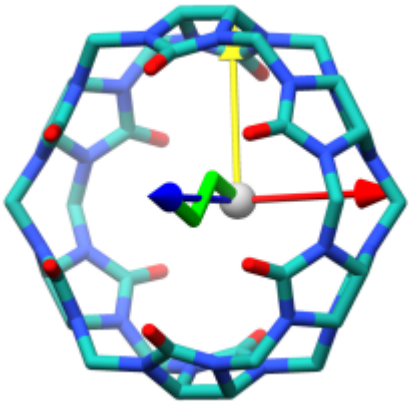
[9]:
```
from paprika.build import align
```

[10]:
```
G1 = ":BUT@C"
G2 = ":BUT@C3"
```

[11]:
```
aligned_structure = align.zalign(structure, G1, G2)
aligned_structure.save("complex/aligned.prmtop", overwrite=True)
aligned_structure.save("complex/aligned.rst7", overwrite=True)
```

```
/home/peanut/pAPRika/paprika/build/align.py:255: RuntimeWarning: invalid value␣
↪encountered in true_divide
  x = np.cross(vector, ref_vector) / np.linalg.norm(np.cross(vector, ref_vector))
```

Here, the origin is shown as a grey sphere, with the z axis drawn as a blue arrow. The coordinates used for this example were already aligned, so Python warns that the cross product is zero, but this won't be the case in general.



Next, we add the dummy atoms. The dummy atoms will be fixed in place during the simulation and are used to orient the host and guest in the lab frame. The dummy atoms are placed along the z axis, behind the host. The dummy atoms are used in distance, angle, and torsion restraints and therefore, the exact positioning of these atoms affects the value of those restraints. For a typical host-guest system, like the one here, we generally place the first dummy atom 6 Angstroms behind the origin, the second dummy atom 9 Angstroms behind the origin, and the third dummy atom 11.2 Angstroms behind the origin and offset about 2.2 Angstroms along the y axis. After we add restraints, the positioning of the dummy atoms should be more clear.

Note, these dummy atoms do not interact with the other atoms in the system, and therefore, there is no problem placing them near host atoms.

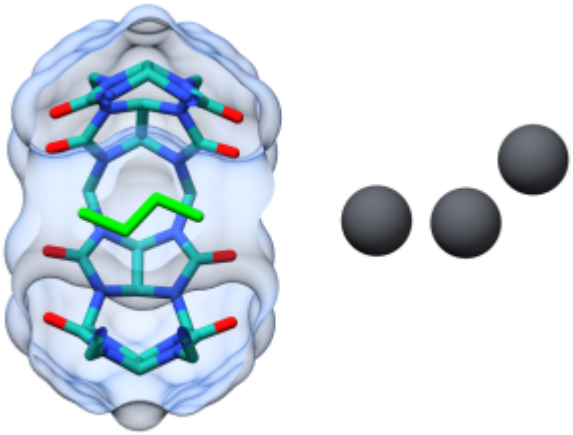
Add dummy atoms

[12]:
```
from paprika.build import dummy
```

[13]:
```
structure = pmd.load_file("complex/aligned.prmtop",
                          "complex/aligned.rst7",
                          structure=True)
```

[14]:
```
structure = dummy.add_dummy(structure, residue_name="DM1", z=-6.0)
structure = dummy.add_dummy(structure, residue_name="DM2", z=-9.0)
structure = dummy.add_dummy(structure, residue_name="DM3", z=-11.2, y=2.2)
```

[15]:
```
structure.save("complex/aligned_with_dummy.prmtop", overwrite=True)
structure.save("complex/aligned_with_dummy.rst7", overwrite=True)
structure.save("complex/aligned_with_dummy.pdb", overwrite=True)
```



When we solvate the system in `tleap`, we will need `frcmod` files for the dummy atoms (otherwise `tleap` will use GAFF parameters and the dummy atoms will *not* be non-interacting). There is a convenient method in `paprika` to write a `frcmod` file that only contains a `MASS` section. For convenience, I am also going to write `mol2` files for each of the dummy atoms. This makes it easy to build up the system, piece-by-piece, if we have a separate `mol2` file for each component of the system: host, guest, dummy atoms.

In principle, an APR calculate can be completed without dummy atoms, by simply lengthening the distance between the host and guest, but the addition of dummy atoms permits an easier way to think about dissociating the guest from the host. (Also, in the absence of dummy atoms, it is challenging to pull the guest straight out of the cavity without adding additional restraints.)

[16]:
```
dummy.write_dummy_frcmod(filepath="complex/dummy.frcmod")
dummy.write_dummy_mol2(residue_name="DM1", filepath="complex/dm1.mol2")
dummy.write_dummy_mol2(residue_name="DM2", filepath="complex/dm2.mol2")
dummy.write_dummy_mol2(residue_name="DM3", filepath="complex/dm3.mol2")
```

Now all the pieces are in place to build the system for an APR calculation.

Put it all together

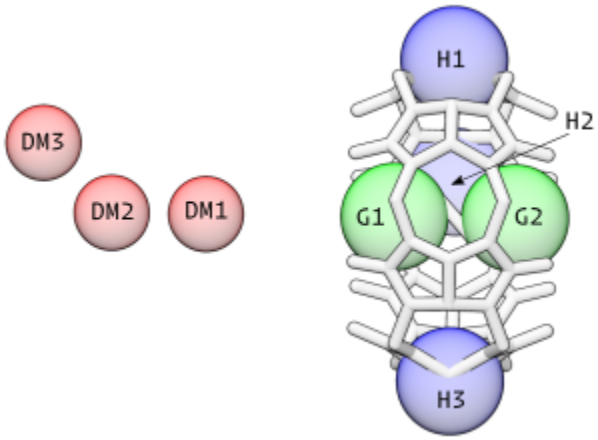
[17]:
```
system = TLeap()
system.output_path = "complex"
system.pbc_type = None
system.neutralize = False

system.template_lines = [
    "source leaprc.gaff",
    f"loadamberparams {data}/cb6.frcmod",
    "loadamberparams dummy.frcmod",
    f"CB6 = loadmol2 {data}/cb6.mol2",
    f"BUT = loadmol2 {data}/but.mol2",
    "DM1 = loadmol2 dm1.mol2",
    "DM2 = loadmol2 dm2.mol2",
```

(continues on next page)

(continued from previous page)

```
    "DM3 = loadmol2 dm3.mol2",
    "model = loadpdb aligned_with_dummy.pdb",
    "check model",
    "savepdb model cb6-but-dum.pdb",
    "saveamberparm model cb6-but-dum.prmtop cb6-but-dum.rst7"
]
system.build()
```

Now we have AMBER coordinates and parameters for the `cb6-but` system with dummy atoms in the appropriate place and with the proper "dummy" parametesr.



Determine the number of windows

Before we add the restraints, it is helpful to set the λ fractions that control the strength of the force constants during attach and release, and to define the distances for the pulling phase.

The attach fractions go from 0 to 1 and we place more points at the bottom of the range to sample the curvature of $dU/d\lambda$. Next, we generally apply a distance restraint until the guest is ~18 Angstroms away from the host, in increments of 0.4 Angstroms. This distance should be at least twice the Lennard-Jones cutoff in the system. These values have worked well for us, but this is one aspect that should be carefully checked for new systems.

[14]:
```
attach_string = "0.00 0.40 0.80 1.60 2.40 4.00 5.50 8.65 11.80 18.10 24.40 37.00 49.60␣
↪74.80 100.00"
attach_fractions = [float(i) / 100 for i in attach_string.split()]
```

[15]:
```
import numpy as np
initial_distance = 6.0
pull_distances = np.arange(0.0 + initial_distance, 18.0 + initial_distance, 1.0)
```

These values will be used to measure distance relative to the first dummy atom, hence the addition of `6.00`.

[16]:
```
release_fractions = []
```

Later, I will explain why there are no release windows in this calculation.

[17]:
```
windows = [len(attach_fractions), len(pull_distances), len(release_fractions)]
print(f"There are {windows} windows in this attach-pull-release calculation.")
```

```
There are [15, 18, 0] windows in this attach-pull-release calculation.
```

Alternatively, we could specify the number of windows for each phase and the force constants and targets will be linearly interpolated. Other ways of specifying these values are documented in the code.

5.3.2 Add restraints using pAPRika

For the host-guest complex, we will apply four different types ot restraints in `pAPRika`:

- Static restraints: these six restraints keep the host and in the proper orientation during the simulation (necessary),
- Guest restraints: these restraints pull the guest away from the host along the z axis (necessary),
- Conformational restraints: these restraints alter the conformational sampling of the host molecule (optional), and
- Wall restraints: these restraints help define the bound state of the guest (optional).

More information on these restraints can be found in:

Henriksen, N.M., Fenley, A.T., and Gilson, M.K. (2015). Computational Calorimetry: High-Precision Calculation of Host-Guest Binding Thermodynamics. J. Chem. Theory Comput. 11, 4377–4394. DOI

In this example, I will show how to setup the static restraints and the guest restraints.

We have already added the dummy atoms and we have already defined the guest anchor atoms. Now we need to define the host anchor atoms (H1, H2, and H3) in the above diagram. The host anchors should be heavy atoms distributed around the cavity (and around the pulling axis). One caveat is that the host anchors should be rigid relative to each other, so conformational restraints do not shift the alignment of the pulling axis relative to the solvation box. For CB6, I have chosen carbons around the central ridge.

[18]:
```
H1 = ":CB6@C"
H2 = ":CB6@C31"
H3 = ":CB6@C18"
```

I'll also make a shorthand for the dummy atoms.

[19]:
```
D1 = ":DM1"
D2 = ":DM2"
D3 = ":DM3"
```

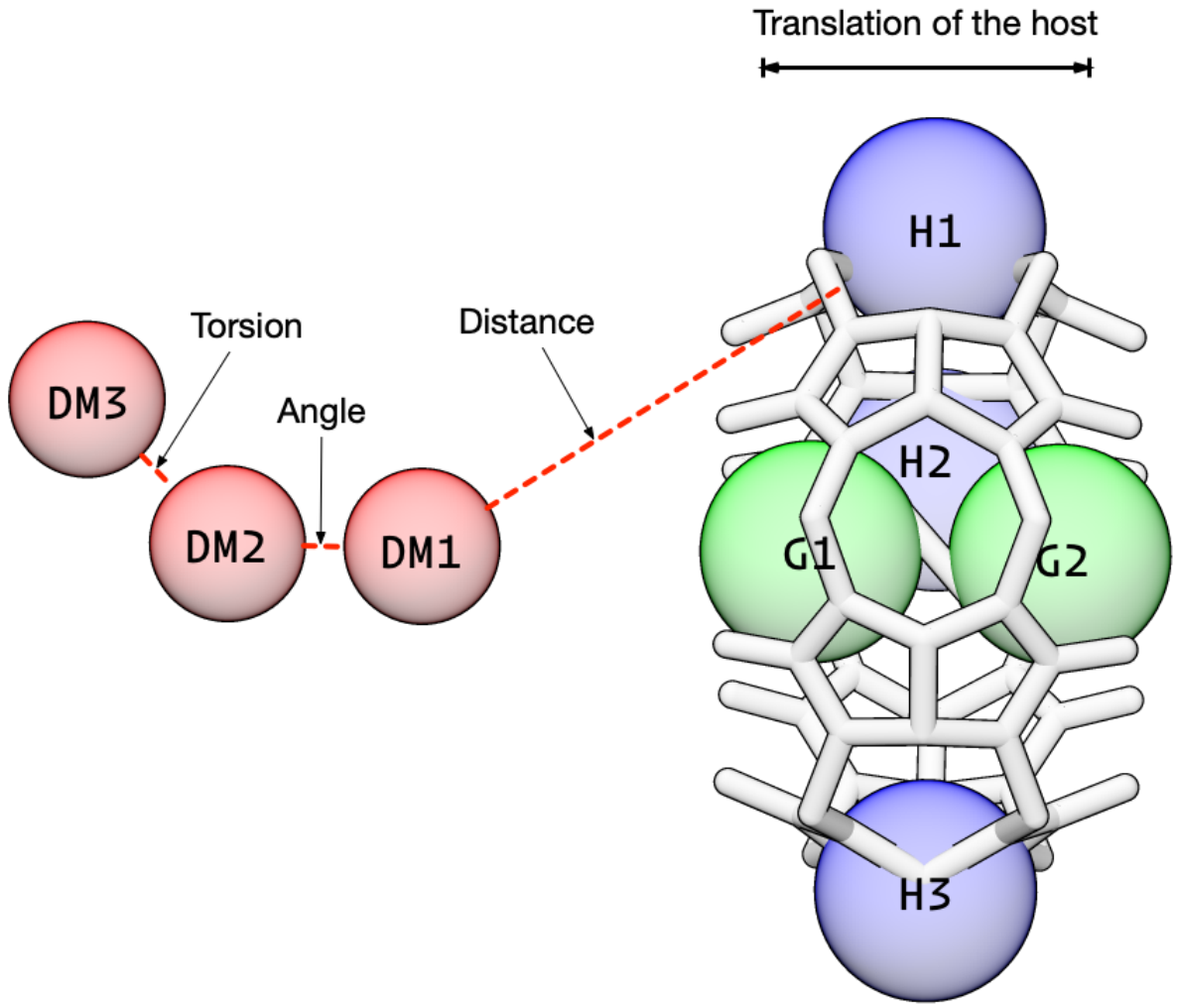
Static restraints

These harmonic restraints are constant throughout the entire simulation. These restraints are used to control the distances and angles between the host and guest relative to the dummy atom. We have created a special class for these restrains, `static_DAT_restraint`, that uses the initial value as the restraint target (this is why the starting structure should be a reasonable facsimile of the bound state).

Note that these restraints are not "attached" and they don't need to be "released" – their force constants do not change in magnitude.

The first three static restraints affect the translational distance, angle, and torsion angle between the host and the dummy atoms. These control the position of the host, via the first anchor atom, from moving relative to the dummy atoms.

There is no *correct* value for the force constants. From experience, we know that a distance force constant of 5.0 kcal/mol/Angstrom^2 won't nail down the host and yet it also won't wander away. Likewise, we have had good results using 100.0 kcal/mol/radian^2 for the angle force constant.



[20]:
```
from paprika import restraints
static_restraints = []
```

[21]:
```
structure = pmd.load_file("complex/cb6-but-dum.prmtop",
                          "complex/cb6-but-dum.rst7",
                          structure = True
                          )
```

[22]:
```
r = restraints.static_DAT_restraint(restraint_mask_list = [D1, H1],
                                    num_window_list = windows,
                                    ref_structure = structure,
                                    force_constant = 5.0,
                                    amber_index=True)

static_restraints.append(r)
```

[23]:
```
r = restraints.static_DAT_restraint(restraint_mask_list = [D2, D1, H1],
                                    num_window_list = windows,
                                    ref_structure = structure,
```

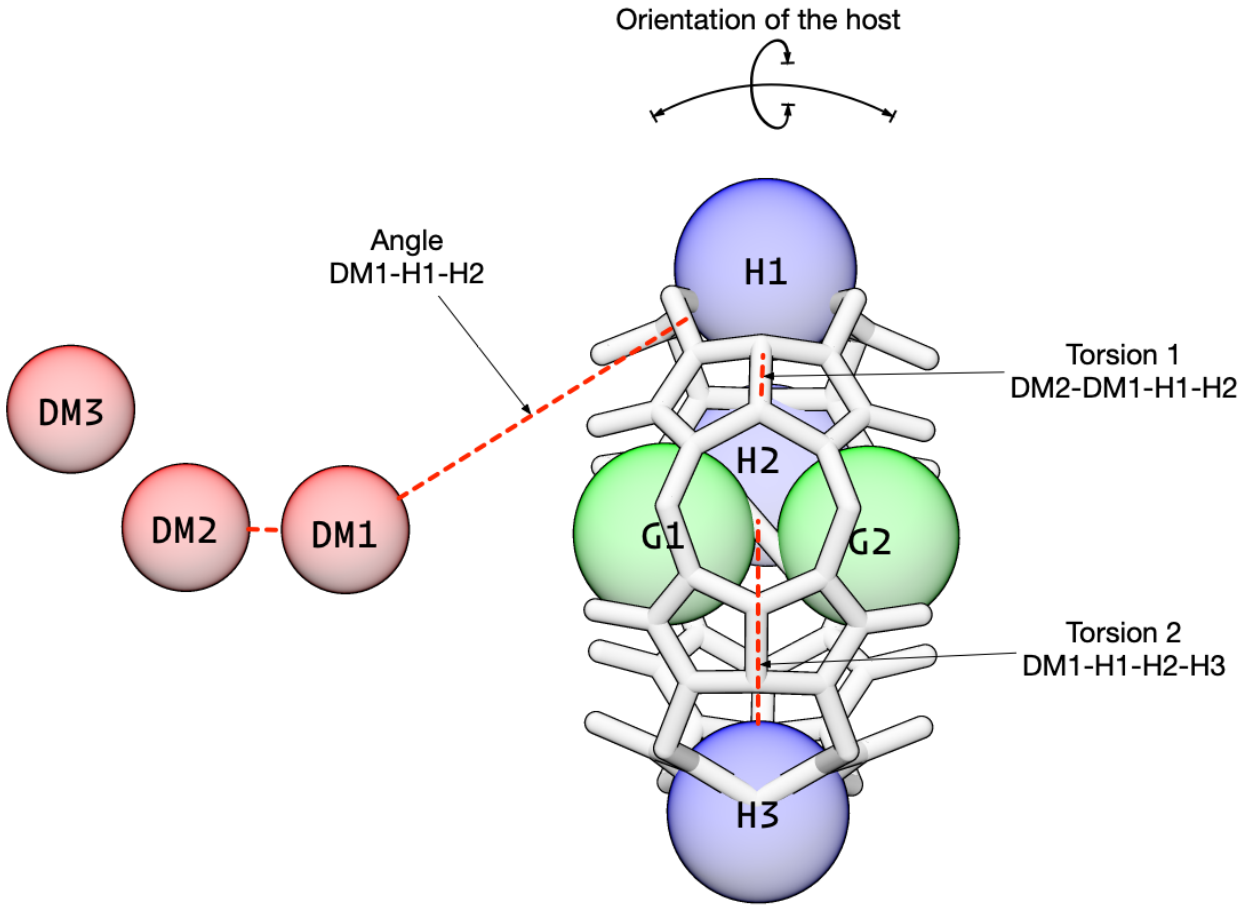
(continues on next page)

(continued from previous page)

```
                                    force_constant = 100.0,
                                    amber_index=True)

static_restraints.append(r)
```

[24]:
```
r = restraints.static_DAT_restraint(restraint_mask_list = [D3, D2, D1, H1],
                                    num_window_list = windows,
                                    ref_structure = structure,
                                    force_constant = 100.0,
                                    amber_index=True)

static_restraints.append(r)
```

The next three restraints control the orientation of the host relative to the dummy atoms. These angle and torsion restraints prevent the host from rotating relative to the dummy atoms.



[25]:
```
r = restraints.static_DAT_restraint(restraint_mask_list = [D1, H1, H2],
                                    num_window_list = windows,
                                    ref_structure = structure,
                                    force_constant = 100.0,
                                    amber_index=True)
```

(continues on next page)

(continued from previous page)

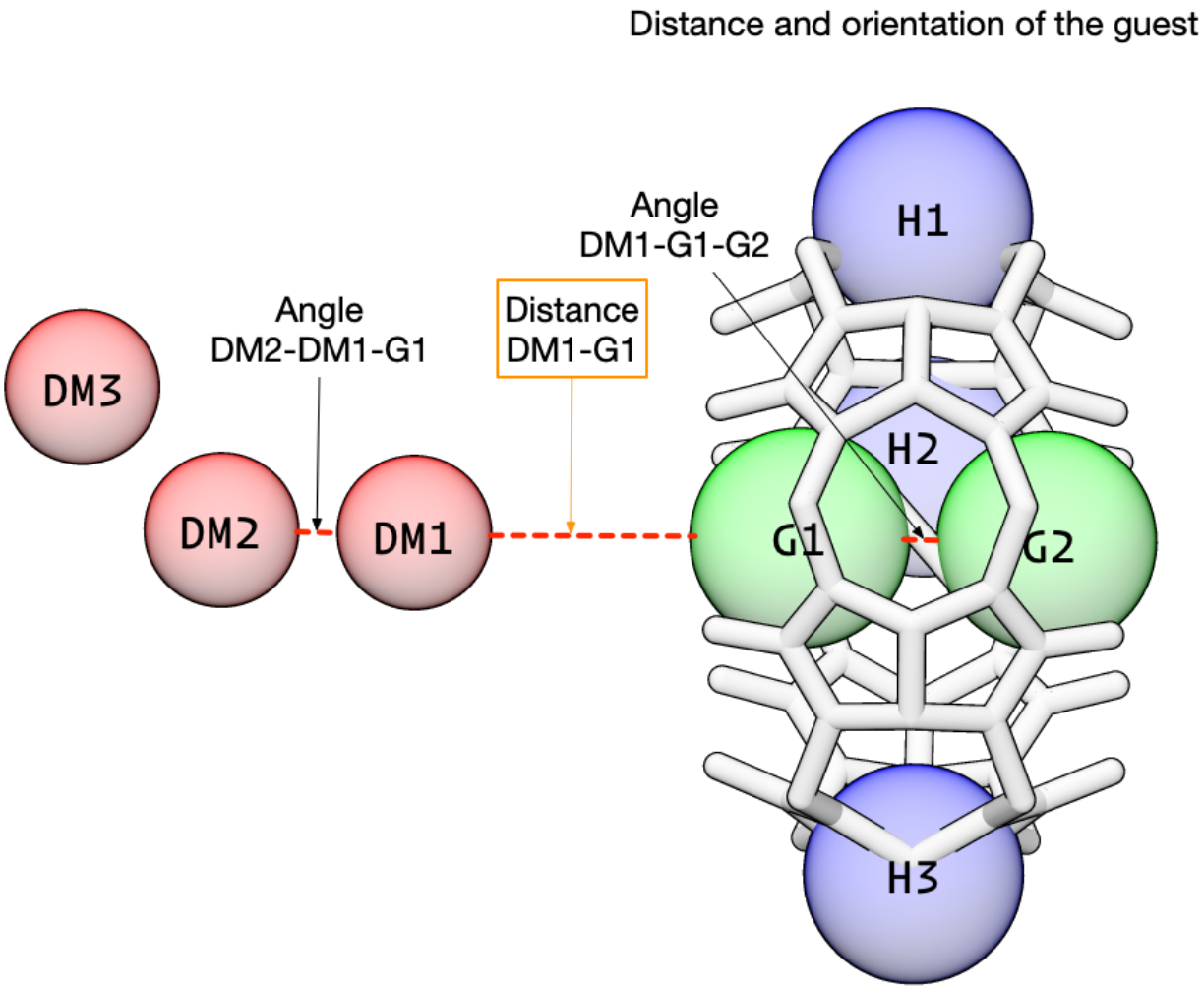
```
static_restraints.append(r)
```

[26]:
```
r = restraints.static_DAT_restraint(restraint_mask_list = [D2, D1, H1, H2],
                                    num_window_list = windows,
                                    ref_structure = structure,
                                    force_constant = 100.0,
                                    amber_index=True)

static_restraints.append(r)
```

[27]:
```
r = restraints.static_DAT_restraint(restraint_mask_list = [D1, H1, H2, H3],
                                    num_window_list = windows,
                                    ref_structure = structure,
                                    force_constant = 100.0,
                                    amber_index=True)

static_restraints.append(r)
```

Guest restraints

Next, we add restraints on the guest. These restraints control the position of the guest and are the key to the attach-pull-release method. During the attach phase, the *force constants* for these restraints is increased from zero. During the pull phase, the *target* for the distance restraint is increased (in the orange box, below), translating the guest away from the host cavity. And during the release phase, the *force constants* are reduced from their "full" value back down to zero.



We use the class `DAT_restraint` to create these three restraints. We will use the same anchor atoms as before, with the same distance and angle force constants. Note that unlike `static_DAT_restraint`, we will first create the restraint, *then* set the attributes, *then* initialize the restraint which does some checks to make sure everything is copacetic.

There are two additional convenience options here:

- `auto_apr = True` sets the force constant during pull to be the final force constant after attach and sets the initial restraint target during pull to be the final attach target.
- `continuous_apr = True` sets the last window of attach to be the same as the first window as pull (and likewise for release)

Also note, due to a quirk with *AMBER*, we specify angle and torsion targets in degrees but the force constant using radians!

[28]:
```
guest_restraints = []
```

[29]:
```
r = restraints.DAT_restraint()
r.mask1 = D1
r.mask2 = G1
r.topology = structure
r.auto_apr = True
```

(continues on next page)

(continued from previous page)

```
r.continuous_apr = True
r.amber_index = True

r.attach["target"] = 6.0                            # Angstroms
r.attach["fraction_list"] = attach_fractions
r.attach["fc_final"] = 5.0                          # kcal/mol/Angstroms**2

r.pull["target_final"] = 24.0                       # Angstroms
r.pull["num_windows"] = windows[1]

r.initialize()
guest_restraints.append(r)
```

[30]:
```
r = restraints.DAT_restraint()
r.mask1 = D2
r.mask2 = D1
r.mask3 = G1
r.topology = structure
r.auto_apr = True
r.continuous_apr = True
r.amber_index = True

r.attach["target"] = 180.0                          # Degrees
r.attach["fraction_list"] = attach_fractions
r.attach["fc_final"] = 100.0                        # kcal/mol/radian**2

r.pull["target_final"] = 180.0                      # Degrees
r.pull["num_windows"] = windows[1]

r.initialize()
guest_restraints.append(r)
```

[31]:
```
r = restraints.DAT_restraint()
r.mask1 = D1
r.mask2 = G1
r.mask3 = G2
r.topology = structure
r.auto_apr = True
r.continuous_apr = True
r.amber_index = True

r.attach["target"] = 180.0                          # Degrees
r.attach["fraction_list"] = attach_fractions
r.attach["fc_final"] = 100.0                        # kcal/mol/radian**2

r.pull["target_final"] = 180.0                      # Degrees
r.pull["num_windows"] = windows[1]

r.initialize()
guest_restraints.append(r)
```

Create the directory structure

We use the guest restraints to create a list of windows with the appropriate names and then we create the directories.

[32]:
```
import os
from paprika.restraints.utils import create_window_list

window_list = create_window_list(guest_restraints)
for window in window_list:
    if not os.path.isdir(f"windows/{window}"):
        os.makedirs(f"windows/{window}")
```

Next, we ask pAPRika to write the restraint information in each window

In each window, we create a file named `disang.rest` to hold all of the restraint information that is required to run the simulation with AMBER. We feed the list of restraints to *pAPRika*, one by one, and it returns the appropriate line.

The functional form of the restraints is specified in section 25.1 of the AMBER18 manual. Specifically, these restraints have a square bottom with parabolic sides out to a specific distance and then linear sides beyond that. The square bottom can be eliminated by setting `r2=r3` and the linear extension can be eliminated by setting the the `r4 = 999` and `r1 = 0`, creating a harmonic restraint

[33]:
```
from paprika.restraints.amber import amber_restraint_line

host_guest_restraints = (static_restraints + guest_restraints)
for window in window_list:
    with open(f"windows/{window}/disang.rest", "w") as file:
        for restraint in host_guest_restraints:
            string = amber_restraint_line(restraint, window)
            if string is not None:
                file.write(string)
```

It is a good idea to open up the `disang.rest` files and see that the force constants and targets make sense (and there are no `NaN` values). Do the force constants for the guest restraints start at zero? Do the targets for the pull slowly increase?

5.3.3 Prepare host-guest system

Translation of the guest

For the attach windows, we will use the initial, bound coordinates for the host-guest complex. Only the force constants change during this phase, so a single set of coordinates is sufficient. For the pull windows, we will translate the guest to the target value of the restraint before solvation, and for the release windows, we will use the coordinates from the final pull window.

[34]:
```
import shutil

for window in window_list:
    if window[0] == "a":
        shutil.copy("complex/cb6-but-dum.prmtop", f"windows/{window}/cb6-but-dum.prmtop")
        shutil.copy("complex/cb6-but-dum.rst7", f"windows/{window}/cb6-but-dum.rst7")
    elif window[0] == "p":
        structure = pmd.load_file("complex/cb6-but-dum.prmtop", "complex/cb6-but-dum.rst7
```

(continues on next page)

(continued from previous page)

```
↪",
                          structure = True)
        target_difference = guest_restraints[0].phase['pull']['targets'][int(window[1:
↪])] - guest_restraints[0].pull['target_initial']
        print(f"In window {window} we will translate the guest {target_difference.
↪magnitude:0.1f}.")
        for atom in structure.atoms:
            if atom.residue.name == "BUT":
                atom.xz += target_difference.magnitude
        structure.save(f"windows/{window}/cb6-but-dum.prmtop", overwrite=True)
        structure.save(f"windows/{window}/cb6-but-dum.rst7", overwrite=True)
```

```
In window p000 we will translate the guest 0.0 Angstroms.
In window p001 we will translate the guest 1.1 Angstroms.
In window p002 we will translate the guest 2.1 Angstroms.
In window p003 we will translate the guest 3.2 Angstroms.
In window p004 we will translate the guest 4.2 Angstroms.
In window p005 we will translate the guest 5.3 Angstroms.
In window p006 we will translate the guest 6.4 Angstroms.
In window p007 we will translate the guest 7.4 Angstroms.
In window p008 we will translate the guest 8.5 Angstroms.
In window p009 we will translate the guest 9.5 Angstroms.
In window p010 we will translate the guest 10.6 Angstroms.
In window p011 we will translate the guest 11.6 Angstroms.
In window p012 we will translate the guest 12.7 Angstroms.
In window p013 we will translate the guest 13.8 Angstroms.
In window p014 we will translate the guest 14.8 Angstroms.
In window p015 we will translate the guest 15.9 Angstroms.
In window p016 we will translate the guest 16.9 Angstroms.
In window p017 we will translate the guest 18.0 Angstroms.
```

Before running a simulation, open each window and check that the position of the host and dummy atoms are fixed and that the position of the guest is bound during the attach windows, moves progressively further during pull, and is away from the host during the release windows.

5.3.4 Simulation

Since we are going to run an implicit solvent simulation, we have everything ready to go. `paprika` has an `amber` module that can help setting default parameters for the simulation. There are some high level options that we set directly, like `simulation.path`, and then we call the function `config_gb_min()` to setup reasonable default simulation parameters for a minimization in the Generalized-Born ensemble. After that, we directly modify the simulation `cntrl` section to apply the positional restraints on the dummy atoms.

For this part, you need to have the *AMBER* executables in your path.

[35]:
```
from paprika.simulate import AMBER
```

I'm using the `logging` module to keep track of time.

[36]:
```
import logging
from importlib import reload
reload(logging)
```

(continues on next page)

(continued from previous page)

```
logger = logging.getLogger()
logging.basicConfig(
    format='%(asctime)s %(message)s',
    datefmt='%Y-%m-%d %I:%M:%S %p',
    level=logging.INFO
)
```

Energy Minimization

Run a quick minimization in every window. Note that we need to specify `simulation.cntrl["ntr"] = 1` to enable the positional restraints on the dummy atoms.

[37]:
```
for window in window_list:
    simulation = AMBER()
    simulation.executable = "sander"

    simulation.path = f"windows/{window}/"
    simulation.prefix = "minimize"

    simulation.topology = "cb6-but-dum.prmtop"
    simulation.coordinates = "cb6-but-dum.rst7"
    simulation.ref = "cb6-but-dum.rst7"
    simulation.restraint_file = "disang.rest"

    simulation.config_gb_min()
    simulation.cntrl["ntr"] = 1
    simulation.cntrl["restraint_wt"] = 50.0
    simulation.cntrl["restraintmask"] = "'@DUM'"

    logging.info(f"Running minimization in window {window}...")
    simulation.run(overwrite=True)
```

```
2020-10-05 12:11:35 PM Running minimization in window a000...
2020-10-05 12:11:39 PM Running minimization in window a001...
2020-10-05 12:11:43 PM Running minimization in window a002...
2020-10-05 12:11:47 PM Running minimization in window a003...
2020-10-05 12:11:51 PM Running minimization in window a004...
2020-10-05 12:11:55 PM Running minimization in window a005...
2020-10-05 12:11:59 PM Running minimization in window a006...
2020-10-05 12:12:04 PM Running minimization in window a007...
2020-10-05 12:12:08 PM Running minimization in window a008...
2020-10-05 12:12:12 PM Running minimization in window a009...
2020-10-05 12:12:16 PM Running minimization in window a010...
2020-10-05 12:12:20 PM Running minimization in window a011...
2020-10-05 12:12:24 PM Running minimization in window a012...
2020-10-05 12:12:28 PM Running minimization in window a013...
2020-10-05 12:12:33 PM Running minimization in window p000...
2020-10-05 12:12:37 PM Running minimization in window p001...
2020-10-05 12:12:41 PM Running minimization in window p002...
2020-10-05 12:12:45 PM Running minimization in window p003...
```

(continues on next page)

(continued from previous page)

```
2020-10-05 12:12:49 PM Running minimization in window p004...
2020-10-05 12:12:53 PM Running minimization in window p005...
2020-10-05 12:12:57 PM Running minimization in window p006...
2020-10-05 12:13:01 PM Running minimization in window p007...
2020-10-05 12:13:05 PM Running minimization in window p008...
2020-10-05 12:13:09 PM Running minimization in window p009...
2020-10-05 12:13:13 PM Running minimization in window p010...
2020-10-05 12:13:17 PM Running minimization in window p011...
2020-10-05 12:13:21 PM Running minimization in window p012...
2020-10-05 12:13:25 PM Running minimization in window p013...
2020-10-05 12:13:29 PM Running minimization in window p014...
2020-10-05 12:13:33 PM Running minimization in window p015...
2020-10-05 12:13:37 PM Running minimization in window p016...
2020-10-05 12:13:41 PM Running minimization in window p017...
```

Production

For simplicity, I am going to skip equilibration and go straight to production!

[38]:
```
for window in window_list:
    simulation = AMBER()
    simulation.executable = "sander"

    simulation.path = f"windows/{window}/"
    simulation.prefix = "production"

    simulation.topology = "cb6-but-dum.prmtop"
    simulation.coordinates = "minimize.rst7"
    simulation.ref = "cb6-but-dum.rst7"
    simulation.restraint_file = "disang.rest"

    simulation.config_gb_md()
    simulation.cntrl["ntr"] = 1
    simulation.cntrl["restraint_wt"] = 50.0
    simulation.cntrl["restraintmask"] = "'@DUM'"

    logging.info(f"Running production in window {window}...")
    simulation.run(overwrite=True)
```

```
2020-10-05 12:13:45 PM Running production in window a000...
2020-10-05 12:13:48 PM Running production in window a001...
2020-10-05 12:13:52 PM Running production in window a002...
2020-10-05 12:13:55 PM Running production in window a003...
2020-10-05 12:13:59 PM Running production in window a004...
2020-10-05 12:14:02 PM Running production in window a005...
2020-10-05 12:14:06 PM Running production in window a006...
2020-10-05 12:14:09 PM Running production in window a007...
2020-10-05 12:14:13 PM Running production in window a008...
2020-10-05 12:14:16 PM Running production in window a009...
2020-10-05 12:14:19 PM Running production in window a010...
2020-10-05 12:14:23 PM Running production in window a011...
```

(continues on next page)

(continued from previous page)

```
2020-10-05 12:14:26 PM Running production in window a012...
2020-10-05 12:14:30 PM Running production in window a013...
2020-10-05 12:14:33 PM Running production in window p000...
2020-10-05 12:14:37 PM Running production in window p001...
2020-10-05 12:14:40 PM Running production in window p002...
2020-10-05 12:14:44 PM Running production in window p003...
2020-10-05 12:14:47 PM Running production in window p004...
2020-10-05 12:14:51 PM Running production in window p005...
2020-10-05 12:14:54 PM Running production in window p006...
2020-10-05 12:14:57 PM Running production in window p007...
2020-10-05 12:15:01 PM Running production in window p008...
2020-10-05 12:15:04 PM Running production in window p009...
2020-10-05 12:15:07 PM Running production in window p010...
2020-10-05 12:15:11 PM Running production in window p011...
2020-10-05 12:15:14 PM Running production in window p012...
2020-10-05 12:15:17 PM Running production in window p013...
2020-10-05 12:15:21 PM Running production in window p014...
2020-10-05 12:15:24 PM Running production in window p015...
2020-10-05 12:15:27 PM Running production in window p016...
2020-10-05 12:15:31 PM Running production in window p017...
```

5.3.5 Analysis

Once the simulation is completed, we can using the `analysis` module to determine the binding free energy. We supply the location of the parameter information, a string or list for the file names (wildcards supported), the location of the windows, and the restraints on the guest.

In this example, we use the method `ti-block` which determines the free energy using **t**hermodynamic **i**integration and then estimates the standard error of the mean at each data point using blocking analysis. Bootstrapping it used to determine the uncertainty of the full thermodynamic integral for each phase.

After running `compute_free_energy()`, a dictionary called `results` will be populated, that contains the free energy and SEM for each phase of the simulation.

[39]:
```
from paprika import analysis
```

[40]:
```
free_energy = analysis.fe_calc()
free_energy.topology = "cb6-but-dum.prmtop"
free_energy.trajectory = 'production*.nc'
free_energy.path = "windows"
free_energy.restraint_list = guest_restraints
free_energy.collect_data()
free_energy.methods = ['ti-block']
free_energy.ti_matrix = "full"
free_energy.boot_cycles = 1000
free_energy.compute_free_energy()
```

But what about release? The guest's rotational and translational degrees of freedom are still restrained releative to the frame of reference of the host. The work to release these restraints is the difference between the chemical potential of this state, and the chemical potentials of the separate host and guest at standard concentration. This can be calculated analytically without doing additional simulation (see equation 8), using the function `compute_ref_state_work`.

[41]:
```
free_energy.compute_ref_state_work([
    guest_restraints[0], guest_restraints[1], None,
    None, guest_restraints[2], None
])

binding_affinity = -1 * (
free_energy.results["attach"]["ti-block"]["fe"] + \
free_energy.results["pull"]["ti-block"]["fe"] + \
free_energy.results["ref_state_work"]
)

sem = np.sqrt(
free_energy.results["attach"]["ti-block"]["sem"]**2 + \
free_energy.results["pull"]["ti-block"]["sem"]**2)
```

[42]:
```
print(f"The binding affinity for butane and cucurbit[6]uril = {binding_affinity.
↪magnitude:0.2f} +/- {sem.magnitude:0.2f} kcal/mol")
```

```
The binding affinity for butane and cucurbit[6]uril = -2.98 +/- 5.82 kcal/mol
```

There is a large uncertainty associated with this calculation because we only simulated for a very short amount of time in each window and we used a large amount of spacing between each window in the pull phase, but the uncertainty will go down with more time.

The experimental value is $-RT\ln(280 * 10^3 M)$ = -7.44 kcal/mol.

5.4 pAPRika tutorial 2 - Explicit Solvent

In this example, we will setup and simulate butane (BUT) as a guest molecule for the host cucurbit[6]uril (CB6).

Unlike the previous tutorial, in this one, we will solvate the host-guest complex in water with additional Na+ and Cl- ions. This tutorial builds off tutorial 1. I've marked new pieces of information with a blue circle, like so:

Since we have a prepared host-guest-dummy setup from the first tutorial, I'm going to skip the initial `tleap` steps and go right into initializing the restraints.

The default mode of *pAPRika* is to detect whether a simulation has completed successfully, and if so, skip that window. This helps when launching mulitple jobs on a cluster or queuing system that might kill jobs after a pre-determined time. That means that running this notebook with the same set of windows as the first tutorial will not re-run the simulations and the results won't change. The easiest thing to do is to rename or delete the existing `windows` directory, if you would like to compare the results between the two methods.

5.4.1 Initial Setup

[ ]:
```
import os
data = "../../../paprika/data/cb6-but"
```

Determine the number of windows

Before we add the restraints, it is helpful to set the λ fractions that control the strength of the force constants during attach and release, and to define the distances for the pulling phase.

The attach fractions go from 0 to 1 and we place more points at the bottom of the range to sample the curvature of $dU/d\lambda$. Next, we generally apply a distance restraint until the guest is ~18 Angstroms away from the host, in increments of 0.4 Angstroms. This distance should be at least twice the Lennard-Jones cutoff in the system. These values have worked well for us, but this is one aspect that should be carefully checked for new systems.

[1]:
```
attach_string = "0.00 0.40 0.80 1.60 2.40 4.00 5.50 8.65 11.80 18.10 24.40 37.00 49.60␣
↪74.80 100.00"
attach_fractions = [float(i) / 100 for i in attach_string.split()]
```

[2]:
```
import numpy as np
initial_distance = 6.0
pull_distances = np.arange(0.0 + initial_distance, 18.0 + initial_distance, 1.0)
```

These values will be used to measure distance relative to the first dummy atom, hence the addition of `6.00`.

[3]:
```
release_fractions = []
```

[4]:
```
windows = [len(attach_fractions), len(pull_distances), len(release_fractions)]
print(f"There are {windows} windows in this attach-pull-release calculation.")
```

```
There are [15, 18, 0] windows in this attach-pull-release calculation.
```

Alternatively, we could specify the number of windows for each phase and the force constants and targets will be linearly interpolated. Other ways of specifying these values are documented in the code.

5.4.2 Add restraints using pAPRika

`pAPRika` supports four different types of restraints:

- Static restraints: these six restraints keep the host and in the proper orientation during the simulation (necessary),
- Guest restraints: these restraints pull the guest away from the host along the z-axis (necessary),
- Conformational restraints: these restraints alter the conformational sampling of the host molecule (optional), and
- Wall restraints: these restraints help define the bound state of the guest (optional).

More information on these restraints can be found in:

Henriksen, N.M., Fenley, A.T., and Gilson, M.K. (2015). Computational Calorimetry: High-Precision Calculation of Host-Guest Binding Thermodynamics. J. Chem. Theory Comput. 11, 4377–4394. DOI

In this example, I will show how to setup the static restraints and the guest restraints.

We have already added the dummy atoms and we have already defined the guest anchor atoms. Now we need to define the host anchor atoms (H1, H2, and H3) in the above diagram. The host anchors should be heavy atoms distributed around the cavity (and around the pulling axis). One caveat is that the host anchors should be rigid relative to each other, so conformational restraints do not shift the alignment of the pulling axis relative to the solvation box. For CB6, I have chosen carbons around the central ridge.

[5]:
```
G1 = ":BUT@C"
G2 = ":BUT@C3"
```

[6]:
```
H1 = ":CB6@C"
H2 = ":CB6@C31"
H3 = ":CB6@C18"
```

I'll also make a shorthand for the dummy atoms.

[7]:
```
D1 = ":DM1"
D2 = ":DM2"
D3 = ":DM3"
```

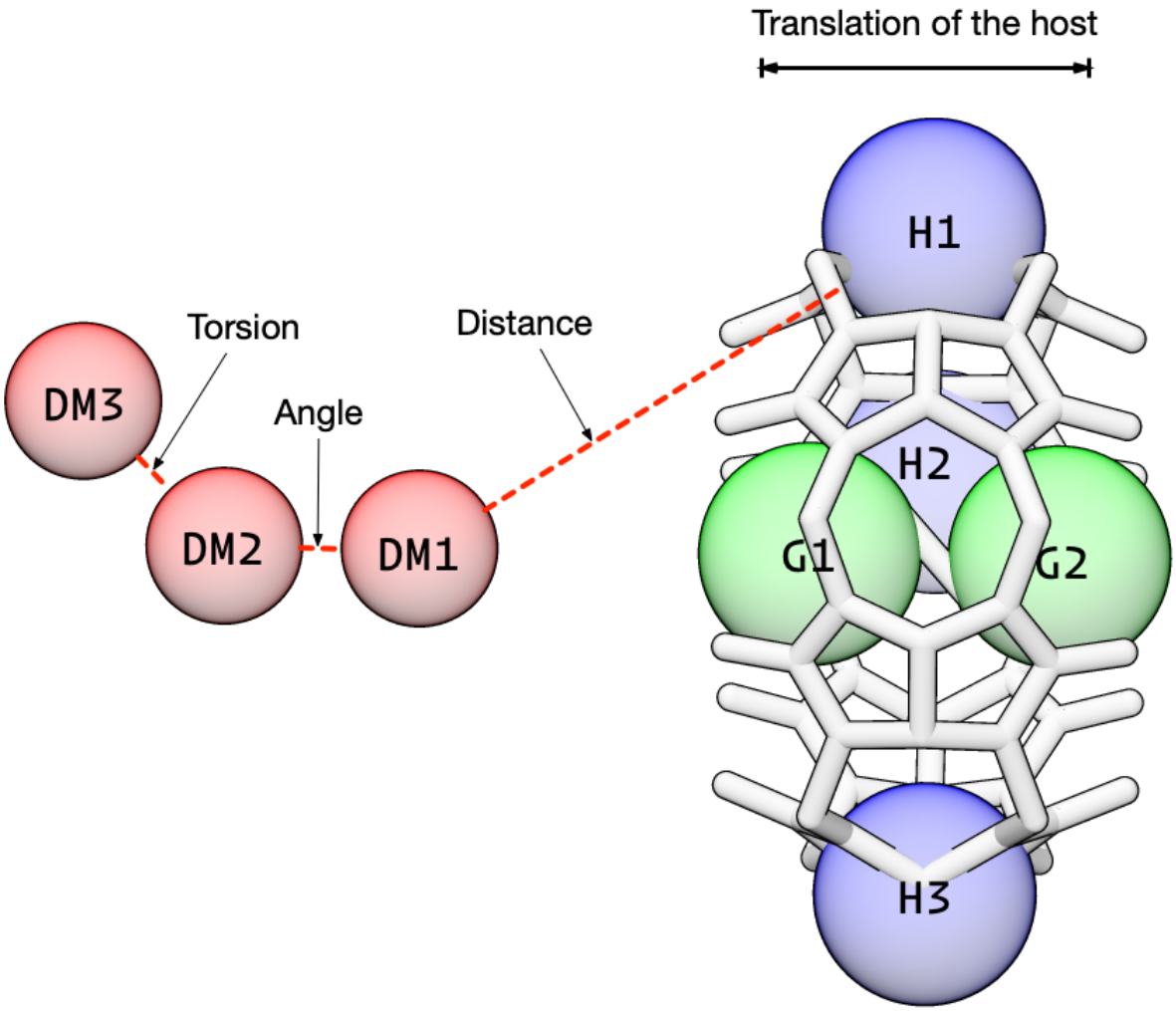
Static restraints

These harmonic restraints are constant throughout the entire simulation. These restraints are used to control the distances and angles between the host and guest relative to the dummy atom. We have created a special class for these restrains, `static_DAT_restraint`, that uses the initial value as the restraint target (this is why the starting structure should be a reasonable facsimile of the bound state).

Note that these restraints are not "attached" and they don't need to be "released" – their force constants do not change in magnitude.

The first three static restraints affect the translational distance, angle, and torsion angle between the host and the dummy atoms. These control the position of the host, via the first anchor atom, from moving relative to the dummy atoms.

There is no *correct* value for the force constants. From experience, we know that a distance force constant of 5.0 kcal/mol/$\mathring{A}^2$ won't nail down the host and yet it also won't wander away. Likewise, we have had good results using 100.0 kcal/mol/rad^2 for the angle force constant.



[8]:
```
from paprika import restraints
static_restraints = []
```

[9]:
```
import parmed as pmd
structure = pmd.load_file("complex/cb6-but-dum.prmtop", "complex/cb6-but-dum.rst7")
```

[10]:
```
r = restraints.static_DAT_restraint(restraint_mask_list = [D1, H1],
                                    num_window_list = windows,
                                    ref_structure = structure,
                                    force_constant = 5.0,
                                    amber_index=True)

static_restraints.append(r)
```

[11]:
```
r = restraints.static_DAT_restraint(restraint_mask_list = [D2, D1, H1],
                                    num_window_list = windows,
                                    ref_structure = structure,
                                    force_constant = 100.0,
                                    amber_index=True)
```

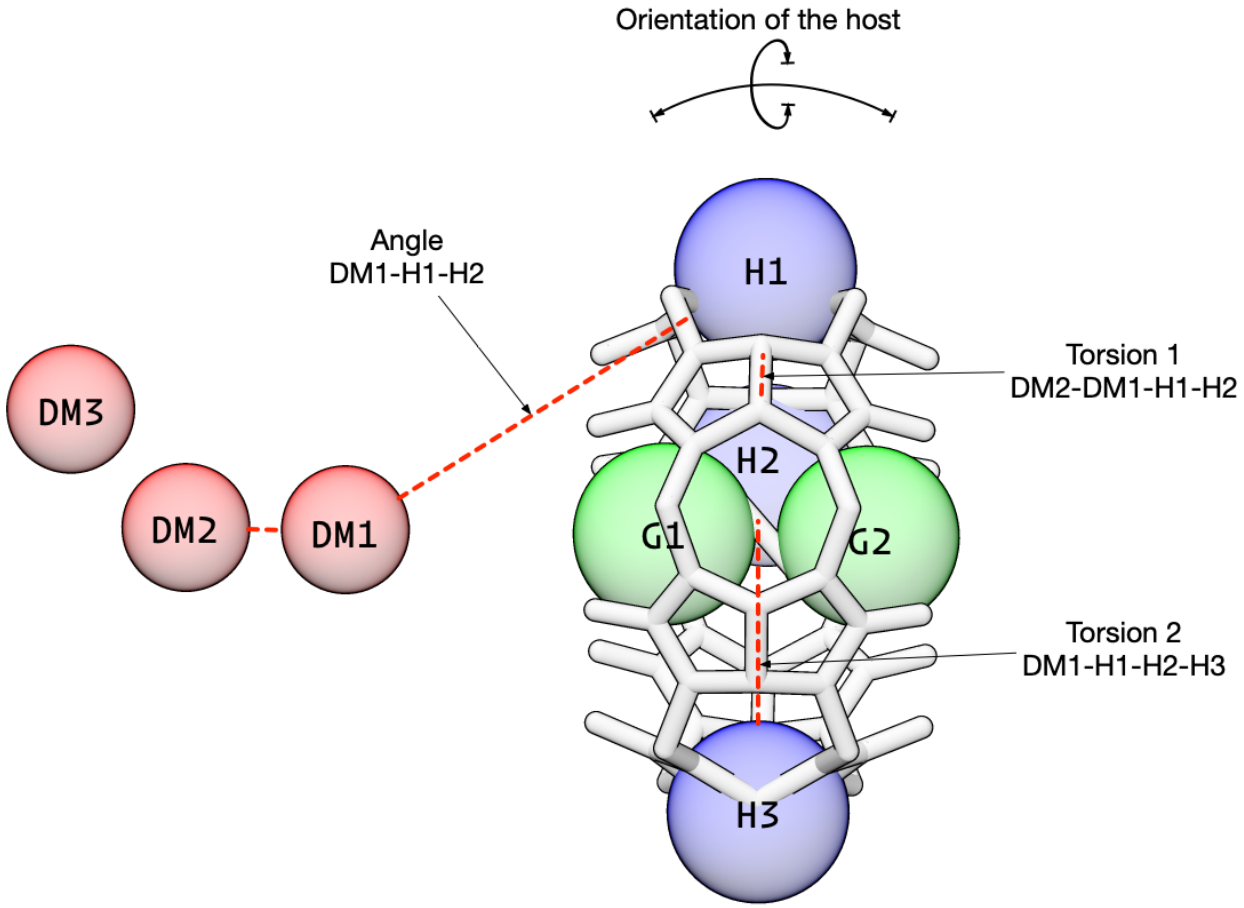
(continues on next page)

(continued from previous page)

```
static_restraints.append(r)
```

[12]:
```
r = restraints.static_DAT_restraint(restraint_mask_list = [D3, D2, D1, H1],
                                    num_window_list = windows,
                                    ref_structure = structure,
                                    force_constant = 100.0,
                                    amber_index=True)

static_restraints.append(r)
```

The next three restraints control the orientation of the host relative to the dummy atoms. These angle and torsion restraints prevent the host from rotating relative to the dummy atoms.



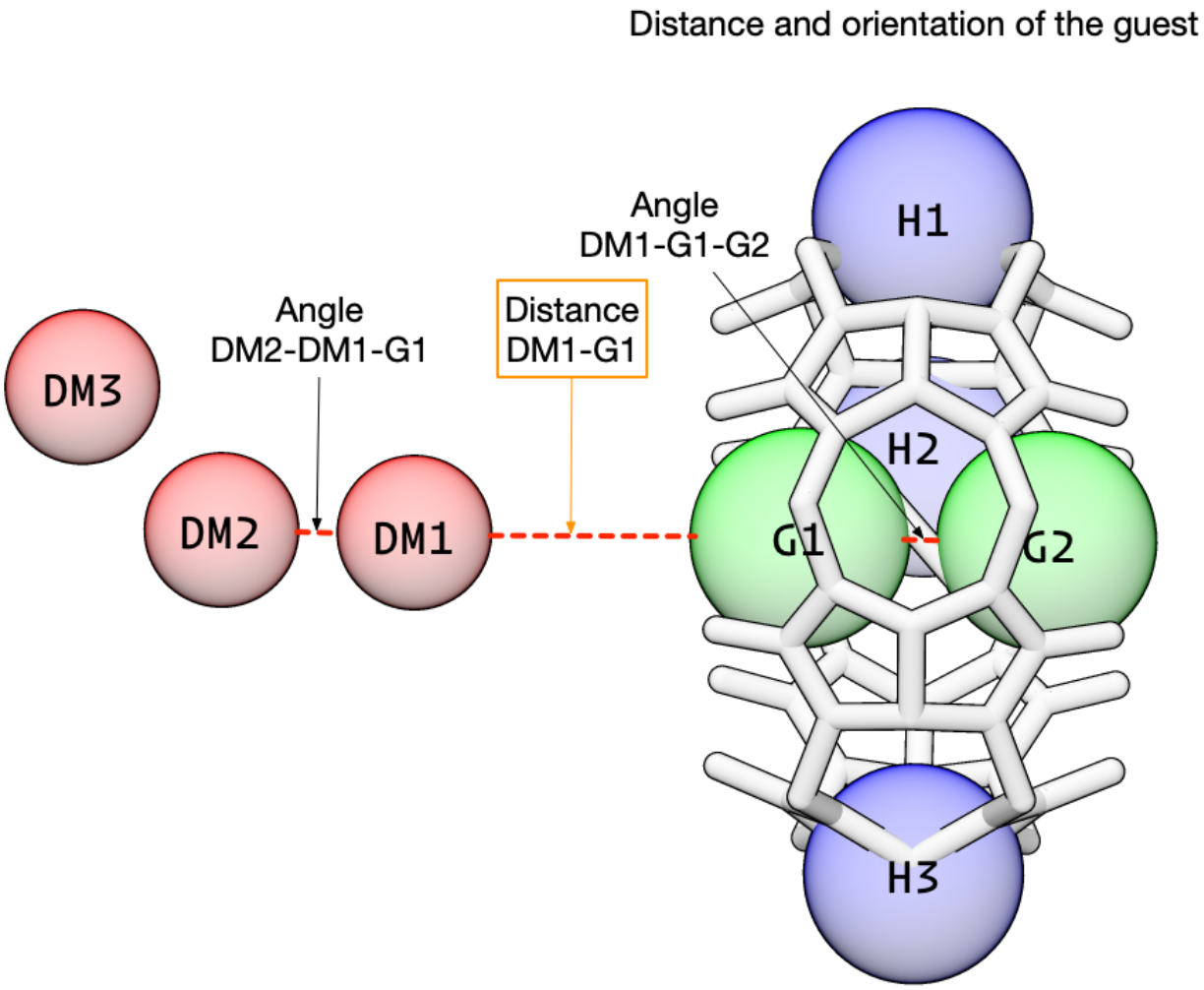
[13]:
```
r = restraints.static_DAT_restraint(restraint_mask_list = [D1, H1, H2],
                                    num_window_list = windows,
                                    ref_structure = structure,
                                    force_constant = 100.0,
                                    amber_index=True)

static_restraints.append(r)
```

[14]:
```
r = restraints.static_DAT_restraint(restraint_mask_list = [D2, D1, H1, H2],
                                    num_window_list = windows,
                                    ref_structure = structure,
                                    force_constant = 100.0,
                                    amber_index=True)

static_restraints.append(r)
```

[15]:
```
r = restraints.static_DAT_restraint(restraint_mask_list = [D1, H1, H2, H3],
                                    num_window_list = windows,
                                    ref_structure = structure,
                                    force_constant = 100.0,
                                    amber_index=True)

static_restraints.append(r)
```

Guest restraints

Next, we add restraints on the guest. These restraints control the position of the guest and are the key to the attach-pull-release method. During the attach phase, the *force constants* for these restraints is increased from zero. During the pull phase, the *target* for the distance restraint is increased (in the orange box, below), translating the guest away from the host cavity. And during the release phase, the *force constants* are reduced from their "full" value back down to zero.



We use the class `DAT_restraint` to create these three restraints. We will use the same anchor atoms as before, with the same distance and angle force constants. Note that unlike `static_DAT_restraint`, we will first create the restraint, *then* set the attributes, *then* initialize the restraint which does some checks to make sure everything is copacetic.

There are two additional convenience options here:

- `auto_apr = True` sets the force constant during pull to be the final force constant after attach and sets the initial restraint target during pull to be the final attach target.
- `continuous_apr = True` sets the last window of attach to be the same as the first window as pull (and likewise for release)

Also note, due to a quirk with AMBER, we specifcy angle and torsion targets in degrees but the force constant using radians!

[16]:
```
guest_restraints = []
```

[17]:
```
r = restraints.DAT_restraint()
r.mask1 = D1
r.mask2 = G1
r.topology = structure
r.auto_apr = True
```

(continues on next page)

(continued from previous page)

```
r.continuous_apr = True
r.amber_index = True

r.attach["target"] = 6.0                            # Angstroms
r.attach["fraction_list"] = attach_fractions
r.attach["fc_final"] = 5.0                          # kcal/mol/Angstroms**2

r.pull["target_final"] = 24.0                       # Angstroms
r.pull["num_windows"] = windows[1]

r.initialize()
guest_restraints.append(r)
```

[18]:
```
r = restraints.DAT_restraint()
r.mask1 = D2
r.mask2 = D1
r.mask3 = G1
r.topology = structure
r.auto_apr = True
r.continuous_apr = True
r.amber_index = True

r.attach["target"] = 180.0                          # Degrees
r.attach["fraction_list"] = attach_fractions
r.attach["fc_final"] = 100.0                        # kcal/mol/radian**2

r.pull["target_final"] = 180.0                      # Degrees
r.pull["num_windows"] = windows[1]

r.initialize()
guest_restraints.append(r)
```

[19]:
```
r = restraints.DAT_restraint()
r.mask1 = D1
r.mask2 = G1
r.mask3 = G2
r.topology = structure
r.auto_apr = True
r.continuous_apr = True
r.amber_index = True

r.attach["target"] = 180.0                          # Degrees
r.attach["fraction_list"] = attach_fractions
r.attach["fc_final"] = 100.0                        # kcal/mol/radian**2

r.pull["target_final"] = 180.0                      # Degrees
r.pull["num_windows"] = windows[1]

r.initialize()
guest_restraints.append(r)
```

Create the directory structure

We use the guest restraints to create a list of windows with the appropriate names and then we create the directories.

[20]:
```
import os
from paprika.restraints.utils import create_window_list

window_list = create_window_list(guest_restraints)
for window in window_list:
    os.makedirs(f"windows/{window}")
```

Next, we ask pAPRika to write the restraint information in each window

In each window, we create a file named `disang.rest` to hold all of the restraint information that is required to run the simulation with AMBER. We feed the list of restraints to pAPRika, one by one, and it returns the appropriate line.

The functional form of the restraints is specified in section 25.1 of the AMBER18 manual. Specifically, these restraints have a square bottom with parabolic sides out to a specific distance and then linear sides beyond that. The square bottom can be eliminated by setting `r2=r3` and the linear extension can be eliminated by setting the the `r4 = 999` and `r1 = 0`, creating a harmonic restraint

[21]:
```
from paprika.restraints.amber import amber_restraint_line

host_guest_restraints = (static_restraints + guest_restraints)
for window in window_list:
    with open(f"windows/{window}/disang.rest", "w") as file:
        for restraint in host_guest_restraints:
            string = amber_restraint_line(restraint, window)
            if string is not None:
                file.write(string)
```

It is a good idea to open up the `disang.rest` files and see that the force constants and targets make sense (and there are no `NaN` values). Do the force constants for the guest restraints start at zero? Do the targets for the pull slowly increase?

Save restraints to a JSON file

The previous code block writes restraints in a format that *AMBER* uses to run simulations in each window, but *pAPRika* uses other information stored in the `DAT_restraint` class in the analysis module. By saving the restraints in a file, it is easy to chunk up setup, simulation, and analysis in different sessions or*python* scripts.

Note: the API for saving and loading restraints is not stable and may be moved into another submodule. The output notes that *pAPRika* is not including the entire topology in the restraint JSON, but will contain a reference to the file that was used to setup th restraints, in this case, `cb6-but-dum.prmtop`.

[22]:
```
from paprika.io import save_restraints
```

[23]:
```
save_restraints(host_guest_restraints, filepath="windows/restraints.json")
```

5.4.3 Prepare host-guest system

Translation of the guest (before solvation)

For the attach windows, we will use the initial, bound coordinates for the host-guest complex. Only the force constants change during this phase, so a single set of coordinates is sufficient. For the pull windows, we will translate the guest to the target value of the restraint before solvation, and for the release windows, we will use the coordinates from the final pull window.

[24]:
```
import shutil

for window in window_list:
    if window[0] == "a":
        shutil.copy("complex/cb6-but-dum.prmtop", f"windows/{window}/cb6-but-dum.prmtop")
        shutil.copy("complex/cb6-but-dum.rst7", f"windows/{window}/cb6-but-dum.rst7")
    elif window[0] == "p":
        structure = pmd.load_file("complex/cb6-but-dum.prmtop", "complex/cb6-but-dum.rst7
↪",
                          structure = True)
        target_difference = guest_restraints[0].phase['pull']['targets'][int(window[1:
↪])] - guest_restraints[0].pull['target_initial']
        print(f"In window {window} we will translate the guest {target_difference.
↪magnitude:0.1f}.")
        for atom in structure.atoms:
            if atom.residue.name == "BUT":
                atom.xz += target_difference.magnitude
        structure.save(f"windows/{window}/cb6-but-dum.prmtop")
        structure.save(f"windows/{window}/cb6-but-dum.rst7")
```

```
In window p000 we will translate the guest 0.0 Angstroms.
In window p001 we will translate the guest 1.1 Angstroms.
In window p002 we will translate the guest 2.1 Angstroms.
In window p003 we will translate the guest 3.2 Angstroms.
In window p004 we will translate the guest 4.2 Angstroms.
In window p005 we will translate the guest 5.3 Angstroms.
In window p006 we will translate the guest 6.4 Angstroms.
In window p007 we will translate the guest 7.4 Angstroms.
In window p008 we will translate the guest 8.5 Angstroms.
In window p009 we will translate the guest 9.5 Angstroms.
In window p010 we will translate the guest 10.6 Angstroms.
In window p011 we will translate the guest 11.6 Angstroms.
In window p012 we will translate the guest 12.7 Angstroms.
In window p013 we will translate the guest 13.8 Angstroms.
In window p014 we will translate the guest 14.8 Angstroms.
In window p015 we will translate the guest 15.9 Angstroms.
In window p016 we will translate the guest 16.9 Angstroms.
In window p017 we will translate the guest 18.0 Angstroms.
```

Before running a simulation, open each window and check that the position of the host and dummy atoms are fixed and that the position of the guest is bound during the attach windows, moves progressively further during pull, and is away from the host during the release windows.

Solvate each window

Unlike the previous tutorial we will solvate the system in each window.

[25]:
```
from paprika.build.system import TLeap
```

By default, this will use cubic periodic boundary conditions; we usually use octahedral periodic boundary conditions to optimize space. Based on experience, 2000 waters is a resaonable amount for CB6, but sometimes it is necessary to run with 2500 or 3000 waters depending on the size of the host and guest molecules. The code below first neutralizes the system (which does not do anything here) and then adds an *additional* 6 Na+ and 6 Cl- ions. The counterion species can be changed via `counter_cation` and `counter_anion`, and the `add_ions` string can take concentrations in the form of `0.150M` or `0.150m` for Molarity or molality, respectively.

Note that when running simulations with periodic boundary conditions, be mindful of the box size relative to the Lennard-Jones cutoff.

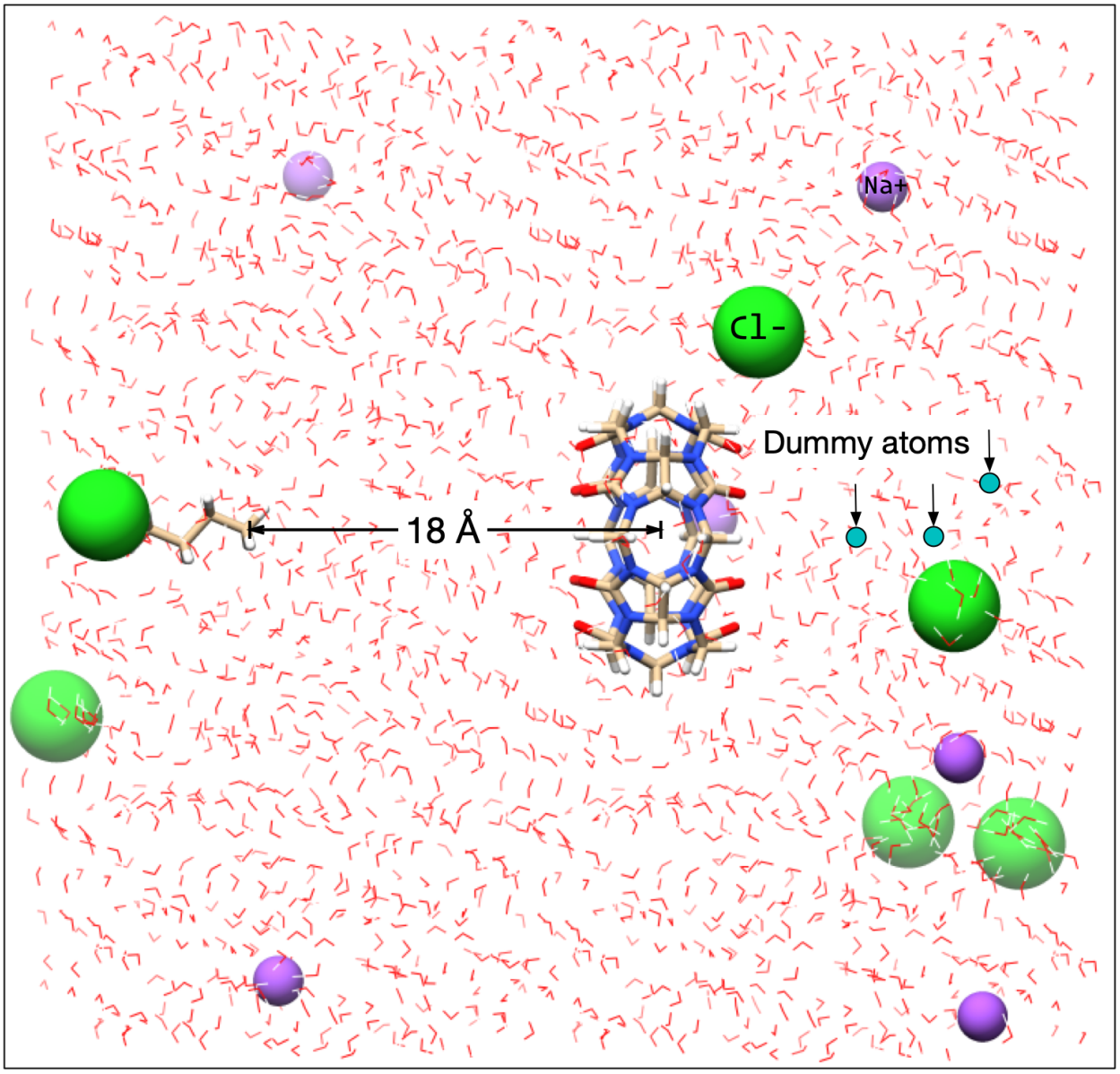
[26]:
```
for window in window_list:
    print(f"Solvating system in window {window}.")
    structure = pmd.load_file(f"windows/{window}/cb6-but-dum.prmtop",
                              f"windows/{window}/cb6-but-dum.rst7")

    if not os.path.exists(f"windows/{window}/cb6-but-dum.pdb"):
        structure.save(f"windows/{window}/cb6-but-dum.pdb")

    system = TLeap()
    system.output_path = os.path.join("windows", window)
    system.output_prefix = "cb6-but-dum-sol"

    system.target_waters = 2000
    system.neutralize = True
    system.add_ions = ["Na+", 6, "Cl-", 6]
    system.template_lines = [
        "source leaprc.gaff",
        "source leaprc.water.tip3p",
        f"loadamberparams {data}/cb6.frcmod",
        "loadamberparams ../../complex/dummy.frcmod",
        f"CB6 = loadmol2 {data}/cb6.mol2",
        f"BUT = loadmol2 {data}/but.mol2",
        "DM1 = loadmol2 ../../complex/dm1.mol2",
        "DM2 = loadmol2 ../../complex/dm2.mol2",
        "DM3 = loadmol2 ../../complex/dm3.mol2",
        "model = loadpdb cb6-but-dum.pdb",
    ]
    system.build()
```

```
Solvating system in window a000.
Solvating system in window a001.
Solvating system in window a002.
Solvating system in window a003.
Solvating system in window a004.
Solvating system in window a005.
Solvating system in window a006.
Solvating system in window a007.
Solvating system in window a008.
Solvating system in window a009.
```

(continues on next page)

(continued from previous page)

```
Solvating system in window a010.
Solvating system in window a011.
Solvating system in window a012.
Solvating system in window a013.
Solvating system in window p000.
Solvating system in window p001.
Solvating system in window p002.
Solvating system in window p003.
Solvating system in window p004.
Solvating system in window p005.
Solvating system in window p006.
Solvating system in window p007.
Solvating system in window p008.
Solvating system in window p009.
Solvating system in window p010.
Solvating system in window p011.
Solvating system in window p012.
Solvating system in window p013.
Solvating system in window p014.
Solvating system in window p015.
Solvating system in window p016.
Solvating system in window p017.
```



5.4.4 Simulation

This is going to look very similar to the the implicit solvent simulation setup, except instead of using `config_gb_min` and `config_gb_sim`, we will use `config_pdb_min` and `config_pbc_sim`.

For this part, you need to have the AMBER executables in your path.

[27]:
```
from paprika.simulate import AMBER
```

Run a quick minimization in every window. Note that we need to specify `simulation.cntrl["ntr"] = 1` to enable the positional restraints on the dummy atoms.

I'm using the `logging` module to keep track of time.

[28]:
```
import logging
from importlib import reload
reload(logging)

logger = logging.getLogger()
logging.basicConfig(
    format='%(asctime)s %(message)s',
    datefmt='%Y-%m-%d %I:%M:%S %p',
    level=logging.INFO
)
```

Energy Minimization

[29]:
```
for window in window_list:
    simulation = AMBER()
    simulation.executable = "mpirun -np 4 pmemd.MPI"

    simulation.path = f"windows/{window}/"
    simulation.prefix = "minimize"

    simulation.topology = "cb6-but-dum-sol.prmtop"
    simulation.coordinates = "cb6-but-dum-sol.rst7"
    simulation.ref = "cb6-but-dum-sol.rst7"
    simulation.restraint_file = "disang.rest"

    simulation.config_pbc_min()
    simulation.cntrl["ntr"] = 1
    simulation.cntrl["restraint_wt"] = 50.0
    simulation.cntrl["restraintmask"] = "'@DUM'"

    logging.info(f"Running minimization in window {window}...")
    simulation.run()
```

```
2020-10-05 12:18:34 PM Running minimization in window a000...
2020-10-05 12:19:00 PM Running minimization in window a001...
2020-10-05 12:19:26 PM Running minimization in window a002...
2020-10-05 12:19:52 PM Running minimization in window a003...
2020-10-05 12:20:18 PM Running minimization in window a004...
2020-10-05 12:20:45 PM Running minimization in window a005...
2020-10-05 12:21:11 PM Running minimization in window a006...
2020-10-05 12:21:37 PM Running minimization in window a007...
2020-10-05 12:22:03 PM Running minimization in window a008...
2020-10-05 12:22:29 PM Running minimization in window a009...
2020-10-05 12:22:56 PM Running minimization in window a010...
2020-10-05 12:23:22 PM Running minimization in window a011...
2020-10-05 12:23:48 PM Running minimization in window a012...
2020-10-05 12:24:14 PM Running minimization in window a013...
2020-10-05 12:24:40 PM Running minimization in window p000...
2020-10-05 12:25:07 PM Running minimization in window p001...
2020-10-05 12:25:33 PM Running minimization in window p002...
2020-10-05 12:25:59 PM Running minimization in window p003...
2020-10-05 12:26:25 PM Running minimization in window p004...
```

(continues on next page)

(continued from previous page)

```
2020-10-05 12:26:51 PM Running minimization in window p005...
2020-10-05 12:27:17 PM Running minimization in window p006...
2020-10-05 12:27:44 PM Running minimization in window p007...
2020-10-05 12:28:10 PM Running minimization in window p008...
2020-10-05 12:28:36 PM Running minimization in window p009...
2020-10-05 12:29:02 PM Running minimization in window p010...
2020-10-05 12:29:28 PM Running minimization in window p011...
2020-10-05 12:29:54 PM Running minimization in window p012...
2020-10-05 12:30:20 PM Running minimization in window p013...
2020-10-05 12:30:47 PM Running minimization in window p014...
2020-10-05 12:31:13 PM Running minimization in window p015...
2020-10-05 12:31:39 PM Running minimization in window p016...
2020-10-05 12:32:05 PM Running minimization in window p017...
```

For simplicity, I am going to skip equilibration and go straight to production!

I'm also going to run a little bit longer in each window, because the additional atoms increases the convergence time. The total simulation time per window, in nanoseconds, is (nstlim) × (dt in fs / 1000) = (50000) × (0.002 / 1000) = 0.1 ns. This is still very short and just for demonstration. A production simulation might require as much as 15 ns per window for binding free energies and up to 1 us per window for binding enthalpies.

Production Run

[33]:
```
for window in window_list:
    simulation = AMBER()
    simulation.executable = "pmemd.cuda"
    simulation.gpu_devices = 0

    simulation.path = f"windows/{window}/"
    simulation.prefix = "production"

    simulation.topology = "cb6-but-dum-sol.prmtop"
    simulation.coordinates = "minimize.rst7"
    simulation.ref = "cb6-but-dum-sol.rst7"
    simulation.restraint_file = "disang.rest"

    simulation.config_pbc_md()
    simulation.cntrl["ntr"] = 1
    simulation.cntrl["restraint_wt"] = 50.0
    simulation.cntrl["restraintmask"] = "'@DUM'"
    simulation.cntrl["nstlim"] = 50000

    logging.info(f"Running production in window {window}...")
    simulation.run(overwrite=True)
```

```
2020-10-05 12:42:12 PM Running production in window a000...
2020-10-05 12:42:25 PM Running production in window a001...
2020-10-05 12:42:38 PM Running production in window a002...
2020-10-05 12:42:51 PM Running production in window a003...
2020-10-05 12:43:04 PM Running production in window a004...
2020-10-05 12:43:18 PM Running production in window a005...
2020-10-05 12:43:31 PM Running production in window a006...
```

(continues on next page)

(continued from previous page)

```
2020-10-05 12:43:44 PM Running production in window a007...
2020-10-05 12:43:57 PM Running production in window a008...
2020-10-05 12:44:11 PM Running production in window a009...
2020-10-05 12:44:24 PM Running production in window a010...
2020-10-05 12:44:37 PM Running production in window a011...
2020-10-05 12:44:50 PM Running production in window a012...
2020-10-05 12:45:04 PM Running production in window a013...
2020-10-05 12:45:17 PM Running production in window p000...
2020-10-05 12:45:30 PM Running production in window p001...
2020-10-05 12:45:43 PM Running production in window p002...
2020-10-05 12:45:57 PM Running production in window p003...
2020-10-05 12:46:10 PM Running production in window p004...
2020-10-05 12:46:23 PM Running production in window p005...
2020-10-05 12:46:36 PM Running production in window p006...
2020-10-05 12:46:50 PM Running production in window p007...
2020-10-05 12:47:03 PM Running production in window p008...
2020-10-05 12:47:16 PM Running production in window p009...
2020-10-05 12:47:29 PM Running production in window p010...
2020-10-05 12:47:43 PM Running production in window p011...
2020-10-05 12:47:56 PM Running production in window p012...
2020-10-05 12:48:09 PM Running production in window p013...
2020-10-05 12:48:22 PM Running production in window p014...
2020-10-05 12:48:35 PM Running production in window p015...
2020-10-05 12:48:49 PM Running production in window p016...
2020-10-05 12:49:02 PM Running production in window p017...
```

5.4.5 Analysis

We will perform the analysis using restraints saved previously to a JSON file.

Once the simulation is completed, we can using the `analysis` module to determine the binding free energy. We supply the location of the parameter information, a string or list for the file names (wildcards supported), the location of the windows, and the restraints on the guest.

In this example, we use the method `ti-block` which determines the free energy using **t**hermodynamic **i**integration and then estimates the standard error of the mean at each data point using blocking analysis. Bootstrapping it used to determine the uncertainty of the full thermodynamic integral for each phase.

After running `compute_free_energy()`, a dictionary called `results` will be populated, that contains the free energy and SEM for each phase of the simulation.

[34]:
```
from paprika.io import load_restraints
```

[35]:
```
guest_restraints = load_restraints("windows/restraints.json")
```

[36]:
```
from paprika import analysis
```

[37]:
```
free_energy = analysis.fe_calc()
free_energy.topology = "cb6-but-dum-sol.prmtop"
free_energy.trajectory = 'production*.nc'
free_energy.path = "windows"
```

(continues on next page)

(continued from previous page)

```
free_energy.restraint_list = guest_restraints
free_energy.collect_data()
free_energy.methods = ['ti-block']
free_energy.ti_matrix = "full"
free_energy.boot_cycles = 1000
free_energy.compute_free_energy()
```

[38]:
```
free_energy.compute_ref_state_work([
    guest_restraints[0], guest_restraints[1], None,
    None, guest_restraints[2], None
])
```

[39]:
```
binding_affinity = -1 * (
free_energy.results["attach"]["ti-block"]["fe"] + \
free_energy.results["pull"]["ti-block"]["fe"] + \
free_energy.results["ref_state_work"]
)

sem = np.sqrt(
free_energy.results["attach"]["ti-block"]["sem"]**2 + \
free_energy.results["pull"]["ti-block"]["sem"]**2
)
```

[42]:
```
print(f"The binding affinity for butane and cucurbit[6]uril = {binding_affinity:0.2f} +/-
↪ {sem:0.2f} kcal/mol")
```

```
The binding affinity for butane and cucurbit[6]uril = -11.62 +/- 1.47 kcal/mol
```

In the first tutorial, using implicit solvation, the binding affinity estimate was -9.00 +/- 5.53 kcal/mol. Now, the affinity estimate is more favorable, by almost 2 kcal/mol, and we've halved the uncertainty. As before, with more windows and more simulation time per window, this value would continue to be refined.

The experimental value is $-RT\ln(280 * 10^3 M)$ = -7.44 kcal/mol.

5.5 pAPRika tutorial 3 - K-Cl dissociation

In this example, we will setup and simulate KCl dissociation using a single distance restraints. This tutorial will demonstrate the use of `pAPRika` besides host-guest systems.

[17]:
```
import os
import json

import numpy as np
import parmed as pmd

from paprika.analysis import fe_calc
from paprika.build.system import TLeap
from paprika.io import NumpyEncoder
from paprika.restraints.restraints import DAT_restraint
from paprika.restraints.amber import amber_restraint_linefrom paprika.restraints.utils␣
↪import create_window_listfrom paprika.simulate import AMBER
```

5.5.1 Specify directory for data

In this case, we just need some initial coordinates. In other cases, we might need `mol2` or `frcmod` files.

[13]:
```
from pathlib import Path
cwd = Path().resolve()
k_cl_pdb = os.path.abspath(os.path.join(cwd, "../../paprika/data/k-cl/k-cl.pdb"))
```

5.5.2 Setup the calculation

Build the vacuum `prmtop` and `inpcrd` files

[18]:
```
# Build the model in vacuum

system = TLeap()
system.template_lines = [
    "source leaprc.water.tip3p",
    "loadamberparams frcmod.ionsjc_tip3p",
    f"model = loadpdb {k_cl_pdb}",
]
system.output_path = "tmp"
system.output_prefix = "k-cl"
system.pbc_type = None
system.target_waters = None
system.neutralize = False
system.build()
```

Specify the number of windows for the umbrella sampling

These are overkill; I have been testing how much data we need to converge this calculation and how quickly we can run a stripped-down version on Travis.

[19]:
```
attach_fractions = np.linspace(0, 1.0, 25)
initial_distance = 2.65
pull_distances = np.linspace(0 + initial_distance, 16.0 + initial_distance, 40)
```

Add a single distance restraint between K+ and Cl-

[20]:
```
restraint = DAT_restraint()
restraint.continuous_apr = True
restraint.amber_index = True
restraint.topology = k_cl_pdb
restraint.mask1 = "@K+"
restraint.mask2 = "@Cl-"

restraint.attach["target"] = initial_distance
restraint.attach["fraction_list"] = attach_fractions
restraint.attach["fc_final"] = 10.0
restraint.pull["fc"] = restraint.attach["fc_final"]
```

(continues on next page)

(continued from previous page)

```
restraint.pull["target_list"] = pull_distances
restraint.initialize()
```

Optionally, add a "wall restraint" to define the bound state and speed convergence

This will apply a harmonic potential that keeps the "guest" Cl- within 3.5 Angstroms.

[21]:
```
wall = DAT_restraint()
wall.auto_apr = False
wall.amber_index = True
wall.topology = k_cl_pdb
wall.mask1 = "@K+"
wall.mask2 = "@Cl-"

wall.attach["fc_initial"] = 1.0
wall.attach["fc_final"] = 1.0

wall.custom_restraint_values["rk2"] = 1.0
wall.custom_restraint_values["rk3"] = 1.0
wall.custom_restraint_values["r1"] = 0.0
wall.custom_restraint_values["r2"] = 3.5
wall.custom_restraint_values["r3"] = 3.5
wall.custom_restraint_values["r4"] = 999

wall.attach["target"] = 3.5
wall.attach["num_windows"] = len(attach_fractions)

wall.initialize()
```

Create the directories for each window and write the AMBER-style restraint input file

This makes it easy to run each window in parallel as a separate simulation.

[22]:
```
# Create folder for the working directory
window_list = create_window_list([restraint])
for window in window_list:
    os.makedirs(f"tmp/windows/{window}", exist_ok=True)

# Write the AMBER restraint files for each window
for window in window_list:
    with open(f"tmp/windows/{window}/disang.rest", "a") as file:
        if window[0] == "a":
            for r in [restraint, wall]:
                string = amber_restraint_line(r, window)
                if string is not None:
                    file.write(string)
        else:
            string = amber_restraint_line(restraint, window)
            file.write(string)
```

(continues on next page)

(continued from previous page)

```
# Generate the topology and coordinates file for each window
for window in window_list:
    if window[0] == "a":
        structure = pmd.load_file("tmp/k-cl.prmtop", "tmp/k-cl.rst7", structure=True)
        for atom in structure.atoms:
            if atom.name == "Cl-":
                atom.xz = 2.65
        structure.save(f"tmp/windows/{window}/k-cl.prmtop", overwrite=True)
        structure.save(f"tmp/windows/{window}/k-cl.rst7", overwrite=True)

    elif window[0] == "p":
        structure = pmd.load_file("tmp/k-cl.prmtop", "tmp/k-cl.rst7", structure=True)
        target_difference = (
            restraint.phase["pull"]["targets"][int(window[1:])]
            - restraint.phase["pull"]["targets"][0]
        )
        print(
            f"In window {window} we will translate the guest {target_difference.
↪magnitude:0.1f}."
        )
        for atom in structure.atoms:
            if atom.name == "Cl-":
                atom.xz += target_difference.magnitude
        structure.save(f"tmp/windows/{window}/k-cl.prmtop", overwrite=True)
        structure.save(f"tmp/windows/{window}/k-cl.rst7", overwrite=True)
```

```
In window p000 we will translate the guest 0.0 Angstroms.
In window p001 we will translate the guest 0.4 Angstroms.
In window p002 we will translate the guest 0.8 Angstroms.
In window p003 we will translate the guest 1.2 Angstroms.
In window p004 we will translate the guest 1.6 Angstroms.
In window p005 we will translate the guest 2.1 Angstroms.
In window p006 we will translate the guest 2.5 Angstroms.
In window p007 we will translate the guest 2.9 Angstroms.
In window p008 we will translate the guest 3.3 Angstroms.
In window p009 we will translate the guest 3.7 Angstroms.
In window p010 we will translate the guest 4.1 Angstroms.
In window p011 we will translate the guest 4.5 Angstroms.
In window p012 we will translate the guest 4.9 Angstroms.
In window p013 we will translate the guest 5.3 Angstroms.
In window p014 we will translate the guest 5.7 Angstroms.
In window p015 we will translate the guest 6.2 Angstroms.
In window p016 we will translate the guest 6.6 Angstroms.
In window p017 we will translate the guest 7.0 Angstroms.
In window p018 we will translate the guest 7.4 Angstroms.
In window p019 we will translate the guest 7.8 Angstroms.
In window p020 we will translate the guest 8.2 Angstroms.
In window p021 we will translate the guest 8.6 Angstroms.
In window p022 we will translate the guest 9.0 Angstroms.
In window p023 we will translate the guest 9.4 Angstroms.
In window p024 we will translate the guest 9.8 Angstroms.
In window p025 we will translate the guest 10.3 Angstroms.
```

(continues on next page)

(continued from previous page)

```
In window p026 we will translate the guest 10.7 Angstroms.
In window p027 we will translate the guest 11.1 Angstroms.
In window p028 we will translate the guest 11.5 Angstroms.
In window p029 we will translate the guest 11.9 Angstroms.
In window p030 we will translate the guest 12.3 Angstroms.
In window p031 we will translate the guest 12.7 Angstroms.
In window p032 we will translate the guest 13.1 Angstroms.
In window p033 we will translate the guest 13.5 Angstroms.
In window p034 we will translate the guest 13.9 Angstroms.
In window p035 we will translate the guest 14.4 Angstroms.
In window p036 we will translate the guest 14.8 Angstroms.
In window p037 we will translate the guest 15.2 Angstroms.
In window p038 we will translate the guest 15.6 Angstroms.
In window p039 we will translate the guest 16.0 Angstroms.
```

Optionally, tweak some parameters, like changing the charge of K+ to 1.3 and Cl- to -1.3

[23]:
```
for window in window_list:
    structure = pmd.load_file(
        f"tmp/windows/{window}/k-cl.prmtop",
        f"tmp/windows/{window}/k-cl.rst7",
        structure=True,
    )
    for atom in structure.atoms:
        if atom.name == "Cl-":
            atom.charge = -1.3
        elif atom.name == "K+":
            atom.charge = 1.3
    structure.save(f"tmp/windows/{window}/k-cl.prmtop", overwrite=True)
    structure.save(f"tmp/windows/{window}/k-cl.rst7", overwrite=True)
```

Solvate the structure in each window to the same number of waters

[24]:
```
for window in window_list:
    print(f"Solvating window {window}...")

    if os.path.exists(f"tmp/windows/{window}/k-cl-sol.prmtop"):
        print("Skipping...")
        continue

    structure = pmd.load_file(
        f"tmp/windows/{window}/k-cl.prmtop", f"tmp/windows/{window}/k-cl.rst7"
    )

    if not os.path.exists(f"tmp/windows/{window}/k-cl.pdb"):
        structure.save(f"tmp/windows/{window}/k-cl.pdb")

    system = TLeap()
    system.output_path = os.path.join("tmp", "windows", window)
```

(continues on next page)

(continued from previous page)

```
    system.output_prefix = "k-cl-sol"

    system.target_waters = 2000
    system.neutralize = False
    system.template_lines = [
        "source leaprc.water.tip3p",
        "model = loadpdb k-cl.pdb"
    ]
    system.build()
```

```
Solvating window a000...
Solvating window a001...
Solvating window a002...
Solvating window a003...
Solvating window a004...
Solvating window a005...
Solvating window a006...
Solvating window a007...
Solvating window a008...
Solvating window a009...
Solvating window a010...
Solvating window a011...
Solvating window a012...
Solvating window a013...
Solvating window a014...
Solvating window a015...
Solvating window a016...
Solvating window a017...
Solvating window a018...
Solvating window a019...
Solvating window a020...
Solvating window a021...
Solvating window a022...
Solvating window a023...
Solvating window p000...
Solvating window p001...
Solvating window p002...
Solvating window p003...
Solvating window p004...
Solvating window p005...
Solvating window p006...
Solvating window p007...
Solvating window p008...
Solvating window p009...
Solvating window p010...
Solvating window p011...
Solvating window p012...
Solvating window p013...
Solvating window p014...
Solvating window p015...
Solvating window p016...
Solvating window p017...
```

(continues on next page)

(continued from previous page)

```
Solvating window p018...
Solvating window p019...
Solvating window p020...
Solvating window p021...
Solvating window p022...
Solvating window p023...
Solvating window p024...
Solvating window p025...
Solvating window p026...
Solvating window p027...
Solvating window p028...
Solvating window p029...
Solvating window p030...
Solvating window p031...
Solvating window p032...
Solvating window p033...
Solvating window p034...
Solvating window p035...
Solvating window p036...
Solvating window p037...
Solvating window p038...
Solvating window p039...
```

5.5.3 Run the calculation

We have a few helper functions – like `config_pbc_min()` and `config_pbc_md()` – that help setup some smart defaults for AMBER. (I'll make a note to work on adding this for the OpenMM side of things.) The simulations can either be run directly, as indicated below, with `simulation.run()` or the input file can be written using `_amber_write_input_file()` and then wrapped using a cluster script (like PBS or whatever).

Energy Minimization

[ ]:
```
for window in window_list:
    simulation = AMBER()
    simulation.executable = "pmemd.cuda"

    simulation.path = f"tmp/windows/{window}/"
    simulation.prefix = "minimize"

    simulation.topology = "k-cl-sol.prmtop"
    simulation.coordinates = "k-cl-sol.rst7"
    simulation.ref = "k-cl-sol.rst7"
    simulation.restraint_file = "disang.rest"

    simulation.config_pbc_min()
    simulation.cntrl["ntr"] = 1
    simulation.cntrl["restraint_wt"] = 50.0
    simulation.cntrl["restraintmask"] = "':1-2'"
    print(f"Running minimization in window {window}...")
    simulation.run()
```

Production Run

```
[ ]: # Simulate
for window in window_list:
    simulation = AMBER()
    simulation.executable = "pmemd.cuda"

    simulation.path = f"tmp/windows/{window}/"
    simulation.prefix = "production"

    simulation.topology = "k-cl-sol.prmtop"
    simulation.coordinate = "minimize.rst7"
    simulation.ref = "k-cl-sol.rst7"
    simulation.restraint_file = "disang.rest"

    simulation.config_pbc_md()
    simulation.cntrl["nstlim"] = 50000

    print(f"Running production in window {window}...")
    simulation.run()
```

5.5.4 Analyze the simulation

Setup the analysis

The analysis needs to know about:

- The parameter file that was used for the molecules,
- The simulation path,
- The trajectories, and
- The method to do the analysis (e.g., TI for the free energy with blocking analysis for the SEM)

```
[ ]: free_energy = fe_calc()
free_energy.topology = "k-cl-sol.prmtop"
free_energy.trajectory = "production*.nc"
free_energy.path = "tmp/windows"
free_energy.restraint_list = [restraint]
free_energy.collect_data()
free_energy.methods = ["ti-block", "mbar-block"]
free_energy.ti_matrix = "full"
free_energy.boot_cycles = 1000
```

Run the analysis and save the results

[ ]:
```
free_energy.compute_free_energy()
free_energy.compute_ref_state_work([restraint, None, None, None, None, None])
```

[ ]:
```
with open("./tmp/results.json", "w") as f:
    dumped = json.dumps(free_energy.results, cls=NumpyEncoder)
    f.write(dumped)

binding_affinity = -1 * (
    free_energy.results["attach"]["ti-block"]["fe"]
    + free_energy.results["pull"]["ti-block"]["fe"]
    + free_energy.results["ref_state_work"]
)

sem = np.sqrt(
    free_energy.results["attach"]["ti-block"]["sem"] ** 2
    + free_energy.results["pull"]["ti-block"]["sem"] ** 2
)

print(
    f"The binding affinity for K+ (+1.3) and Cl- (-1.3) = {binding_affinity.magnitude:0.
↪2f} +/- {sem.magnitude:0.2f} kcal/mol"
)
```

5.6 pAPRika tutorial 4 - APR/OpenMM

In this tutorial, we will perform APR calculations for butane (BUT)–cucurbit[6]uril (CB6). This is a repeat of *tutorial 1* using `OpenMM` instead of `AMBER` as the simulation engine. We will go through the process of converting APR restraints constructed with *pAPRika* and `AMBER` structure files to an `OpenMM` system.

Since we have a prepared the host-guest-dummy setup from the first tutorial, we will skip the initial `tleap` steps and go right into initializing the restraints.

5.6.1 Initialize

[1]:
```
import os
import json
import shutil
import numpy as np
import parmed as pmd
```

Initialize logger

[2]:
```
import logging
from importlib import reload
reload(logging)

logger = logging.getLogger()
logging.basicConfig(
```

(continues on next page)

(continued from previous page)

```
    format='%(asctime)s %(message)s',
    datefmt='%Y-%m-%d %I:%M:%S %p',
    level=logging.INFO
)
```

Define names

We will store the files created in this tutorial in a folder called `openmm` so we don't mix files with the previous tutorial.

[3]:
```
base_name = "cb6-but-dum"
work_dir = "openmm"
complex_dir = "complex"
```

5.6.2 Generate APR Restraints

NOTE: The only difference here is to set `amber_index` to `False` since `OpenMM` atom numbering starts from 0.

Define anchor atoms

See *tutorial 1* for the choice of selection

[4]:
```
# Guest atoms
G1 = ":BUT@C"
G2 = ":BUT@C3"

# Host atoms
H1 = ":CB6@C"
H2 = ":CB6@C31"
H3 = ":CB6@C18"

# Dummy atoms
D1 = ":DM1"
D2 = ":DM2"
D3 = ":DM3"
```

Determine the number of windows

Before we add the restraints, it is helpful to set the λ fractions that control the strength of the force constants during attach and release, and to define the distances for the pulling phase.

The attach fractions go from 0 to 1 and we place more points at the bottom of the range to sample the curvature of $dU/d\lambda$. Next, we generally apply a distance restraint until the guest is ~18 Angstroms away from the host, in increments of 0.4 Angstroms. This distance should be at least twice the Lennard-Jones cutoff in the system. These values have worked well for us, but this is one aspect that should be carefully checked for new systems.

[5]:
```
attach_string = "0.00 0.40 0.80 1.60 2.40 4.00 5.50 8.65 11.80 18.10 24.40 37.00 49.60␣
↪74.80 100.00"
attach_fractions = [float(i) / 100 for i in attach_string.split()]
```

(continues on next page)

(continued from previous page)

```
initial_distance = 6.0
pull_distances = np.arange(0.0 + initial_distance, 18.0 + initial_distance, 1.0)

release_fractions = []

windows = [len(attach_fractions), len(pull_distances), len(release_fractions)]
logging.info(f"There are {windows} windows in this attach-pull-release calculation.")
```

```
2020-08-19 10:30:19 PM There are [15, 18, 0] windows in this attach-pull-release␣
↪calculation.
```

Load complex structure

[7]:
```
structure = pmd.load_file(
    os.path.join(complex_dir, f"{base_name}.prmtop"),
    os.path.join(complex_dir, f"{base_name}.rst7"),
    structure = True,
)
```

Host Static Restraints

See tutorial 1 for an explanation of the static restraints

[ ]:
```
import paprika.restraints as restraints
```

[8]:
```
static_restraints = []
```

[9]:
```
r = restraints.static_DAT_restraint(restraint_mask_list = [D1, H1],
                                    num_window_list = windows,
                                    ref_structure = structure,
                                    force_constant = 5.0,
                                    amber_index=False)

static_restraints.append(r)
```

[10]:
```
r = restraints.static_DAT_restraint(restraint_mask_list = [D2, D1, H1],
                                    num_window_list = windows,
                                    ref_structure = structure,
                                    force_constant = 100.0,
                                    amber_index=False)

static_restraints.append(r)
```

[11]:
```
r = restraints.static_DAT_restraint(restraint_mask_list = [D3, D2, D1, H1],
                                    num_window_list = windows,
                                    ref_structure = structure,
                                    force_constant = 100.0,
                                    amber_index=False)
```

(continues on next page)

(continued from previous page)

```
static_restraints.append(r)
```

[12]:
```
r = restraints.static_DAT_restraint(restraint_mask_list = [D1, H1, H2],
                                    num_window_list = windows,
                                    ref_structure = structure,
                                    force_constant = 100.0,
                                    amber_index=False)

static_restraints.append(r)
```

[13]:
```
r = restraints.static_DAT_restraint(restraint_mask_list = [D2, D1, H1, H2],
                                    num_window_list = windows,
                                    ref_structure = structure,
                                    force_constant = 100.0,
                                    amber_index=False)

static_restraints.append(r)
```

[14]:
```
r = restraints.static_DAT_restraint(restraint_mask_list = [D1, H1, H2, H3],
                                    num_window_list = windows,
                                    ref_structure = structure,
                                    force_constant = 100.0,
                                    amber_index=False)

static_restraints.append(r)
```

Guest translational and rotational restraints

See tutorial 1 for an explanation of the guest restraints

[15]:
```
guest_restraints = []
```

[16]:
```
r = restraints.DAT_restraint()
r.mask1 = D1
r.mask2 = G1
r.topology = structure
r.auto_apr = True
r.continuous_apr = True
r.amber_index = False

r.attach["target"] = pull_distances[0]          # Angstroms
r.attach["fraction_list"] = attach_fractions
r.attach["fc_final"] = 5.0                      # kcal/mol/Angstroms**2

r.pull["target_final"] = 24.0                   # Angstroms
r.pull["num_windows"] = windows[1]

r.initialize()
guest_restraints.append(r)
```

[17]:
```
r = restraints.DAT_restraint()
r.mask1 = D2
r.mask2 = D1
r.mask3 = G1
r.topology = structure
r.auto_apr = True
r.continuous_apr = True
r.amber_index = False

r.attach["target"] = 180.0                      # Degrees
r.attach["fraction_list"] = attach_fractions
r.attach["fc_final"] = 100.0                    # kcal/mol/radian**2

r.pull["target_final"] = 180.0                  # Degrees
r.pull["num_windows"] = windows[1]

r.initialize()
guest_restraints.append(r)
```

[18]:
```
r = restraints.DAT_restraint()
r.mask1 = D1
r.mask2 = G1
r.mask3 = G2
r.topology = structure
r.auto_apr = True
r.continuous_apr = True
r.amber_index = False

r.attach["target"] = 180.0                      # Degrees
r.attach["fraction_list"] = attach_fractions
r.attach["fc_final"] = 100.0                    # kcal/mol/radian**2

r.pull["target_final"] = 180.0                  # Degrees
r.pull["num_windows"] = windows[1]

r.initialize()
guest_restraints.append(r)
```

Create APR windows

We use the guest restraints to create a list of windows with the appropriate names and then create the directories.

[19]:
```
from paprika.restraints.utils import create_window_list
```

[20]:
```
window_list = create_window_list(guest_restraints)
```

```
2020-08-19 10:30:19 PM Restraints appear to be consistent
```

[21]:
```
for window in window_list:
    folder = os.path.join(work_dir, window)
    if not os.path.isdir(folder):
        os.makedirs(os.path.join(work_dir, window))
```

5.6.3 Prepare host-guest system

Translate guest molecule

For the attach windows, we will use the initial, bound coordinates for the host-guest complex. Only the force constants change during this phase, so a single set of coordinates is sufficient. For the pull windows, we will translate the guest to the target value of the restraint before solvation, and for the release windows, we will use the coordinates from the final pull window.

[22]:
```
for window in window_list:
    if window[0] == "a":
        shutil.copy(os.path.join(complex_dir, f"{base_name}.prmtop"),
                    os.path.join(work_dir, window, f"{base_name}.prmtop"))
        shutil.copy(os.path.join(complex_dir, f"{base_name}.rst7"),
                    os.path.join(work_dir, window, f"{base_name}.rst7"))

    elif window[0] == "p":
        structure = pmd.load_file(
            os.path.join(complex_dir, f"{base_name}.prmtop"),
            os.path.join(complex_dir, f"{base_name}.rst7"),
            structure = True
        )
        target_difference = guest_restraints[0].phase['pull']['targets'][int(window[1:
↪])] -\
                            guest_restraints[0].pull['target_initial']
        logging.info(f"In window {window} we will translate the guest {target_difference.
↪magnitude:0.1f}.")

        for atom in structure.atoms:
            if atom.residue.name == "BUT":
                atom.xz += target_difference.magnitude

        structure.save(os.path.join(work_dir, window, f"{base_name}.prmtop"),␣
↪overwrite=True)
        structure.save(os.path.join(work_dir, window, f"{base_name}.rst7"),␣
↪overwrite=True)
```

```
2020-08-19 10:30:19 PM In window p000 we will translate the guest 0.0 Angstroms.
2020-08-19 10:30:19 PM In window p001 we will translate the guest 1.1 Angstroms.
2020-08-19 10:30:19 PM In window p002 we will translate the guest 2.1 Angstroms.
2020-08-19 10:30:19 PM In window p003 we will translate the guest 3.2 Angstroms.
2020-08-19 10:30:19 PM In window p004 we will translate the guest 4.2 Angstroms.
2020-08-19 10:30:19 PM In window p005 we will translate the guest 5.3 Angstroms.
2020-08-19 10:30:19 PM In window p006 we will translate the guest 6.4 Angstroms.
2020-08-19 10:30:19 PM In window p007 we will translate the guest 7.4 Angstroms.
2020-08-19 10:30:19 PM In window p008 we will translate the guest 8.5 Angstroms.
2020-08-19 10:30:19 PM In window p009 we will translate the guest 9.5 Angstroms.
2020-08-19 10:30:19 PM In window p010 we will translate the guest 10.6 Angstroms.
2020-08-19 10:30:19 PM In window p011 we will translate the guest 11.6 Angstroms.
2020-08-19 10:30:19 PM In window p012 we will translate the guest 12.7 Angstroms.
2020-08-19 10:30:19 PM In window p013 we will translate the guest 13.8 Angstroms.
2020-08-19 10:30:19 PM In window p014 we will translate the guest 14.8 Angstroms.
2020-08-19 10:30:19 PM In window p015 we will translate the guest 15.9 Angstroms.
2020-08-19 10:30:19 PM In window p016 we will translate the guest 16.9 Angstroms.
```

(continues on next page)

(continued from previous page)

```
2020-08-19 10:30:19 PM In window p017 we will translate the guest 18.0 Angstroms.
```

Create OpenMM system and apply restraints

Here, we will convert the *AMBER* `*.prmtop` & `*.rst7` to an *OpenMM* system object for each window and convert it to a *XML* file. The Generalized Born Implicit Solvent model we will use is `HCT`, which is equivalent to `igb=1` in *AMBER*. Afterwords, we will apply restraints on the dummy atoms using `apply_positional_restraints` and the static & guest restraints with `apply_dat_restraint`.

[23]:
```
import openmm.unit as unit
import openmm.app as app
import openmm as openmm

from paprika.restraints.utils import parse_window
from paprika.restraints.openmm import apply_positional_restraints, apply_dat_restraint
```

[24]:
```
for window in window_list:
    # Current window
    folder = os.path.join(work_dir, window)
    window_number, phase = parse_window(window)
    logging.info(f"Creating XML for in window {window}")

    # Load Amber
    prmtop = app.AmberPrmtopFile(os.path.join(folder, f'{base_name}.prmtop'))
    inpcrd = app.AmberInpcrdFile(os.path.join(folder, f'{base_name}.rst7'))

    # Create PDB file
    with open(os.path.join(folder, 'system.pdb'), 'w') as file:
        app.PDBFile.writeFile(prmtop.topology, inpcrd.positions, file, keepIds=True)

    # Create an OpenMM system from the Amber topology
    system = prmtop.createSystem(
        nonbondedMethod=app.NoCutoff,
        constraints=app.HBonds,
        implicitSolvent=app.HCT,
    )

    # Apply positional restraints on the dummy atoms
    apply_positional_restraints(os.path.join(folder, 'system.pdb'), system, force_
↪group=15)

    # Apply host static restraints
    for restraint in static_restraints:
        apply_dat_restraint(system, restraint, phase, window_number, force_group=10)

    # Apply guest restraints
    for restraint in guest_restraints:
        apply_dat_restraint(system, restraint, phase, window_number, force_group=11)

    # Save OpenMM system to XML file
    system_xml = openmm.XmlSerializer.serialize(system)
```

(continues on next page)

(continued from previous page)

```
    with open(os.path.join(folder, 'system.xml'), 'w') as file:
        file.write(system_xml)
```

```
2020-08-19 10:30:19 PM Creating XML for in window a000
2020-08-19 10:30:19 PM Creating XML for in window a001
2020-08-19 10:30:19 PM Creating XML for in window a002
2020-08-19 10:30:19 PM Creating XML for in window a003
2020-08-19 10:30:19 PM Creating XML for in window a004
2020-08-19 10:30:19 PM Creating XML for in window a005
2020-08-19 10:30:19 PM Creating XML for in window a006
2020-08-19 10:30:20 PM Creating XML for in window a007
2020-08-19 10:30:20 PM Creating XML for in window a008
2020-08-19 10:30:20 PM Creating XML for in window a009
2020-08-19 10:30:20 PM Creating XML for in window a010
2020-08-19 10:30:20 PM Creating XML for in window a011
2020-08-19 10:30:20 PM Creating XML for in window a012
2020-08-19 10:30:20 PM Creating XML for in window a013
2020-08-19 10:30:20 PM Creating XML for in window p000
2020-08-19 10:30:20 PM Creating XML for in window p001
2020-08-19 10:30:20 PM Creating XML for in window p002
2020-08-19 10:30:20 PM Creating XML for in window p003
2020-08-19 10:30:20 PM Creating XML for in window p004
2020-08-19 10:30:20 PM Creating XML for in window p005
2020-08-19 10:30:20 PM Creating XML for in window p006
2020-08-19 10:30:20 PM Creating XML for in window p007
2020-08-19 10:30:20 PM Creating XML for in window p008
2020-08-19 10:30:20 PM Creating XML for in window p009
2020-08-19 10:30:20 PM Creating XML for in window p010
2020-08-19 10:30:20 PM Creating XML for in window p011
2020-08-19 10:30:20 PM Creating XML for in window p012
2020-08-19 10:30:20 PM Creating XML for in window p013
2020-08-19 10:30:20 PM Creating XML for in window p014
2020-08-19 10:30:20 PM Creating XML for in window p015
2020-08-19 10:30:20 PM Creating XML for in window p016
2020-08-19 10:30:20 PM Creating XML for in window p017
```

5.6.4 Simulation

For this part, you need to have `OpenMM` installed and by default the simulations will run on the CPU. See the `OpenMM` documentation if you want to run the simulation on the GPU. We will set the integrator time step to 1 fs with a total of 50,000 steps for production run and the temperature is set to 300 K.

Energy Minimization

[25]:
```
for window in window_list:
    folder = os.path.join(work_dir, window)
    logging.info(f"Running minimization in window {window}...")

    # Load XML and input coordinates
    with open(os.path.join(folder, 'system.xml'), 'r') as file:
        system = openmm.XmlSerializer.deserialize(file.read())
    coords = app.PDBFile(os.path.join(folder, 'system.pdb'))

    # Integrator
    integrator = openmm.LangevinIntegrator(300 * unit.kelvin, 1.0 / unit.picoseconds, 1.
↪0 * unit.femtoseconds)

    # Simulation Object
    simulation = app.Simulation(coords.topology, system, integrator)
    simulation.context.setPositions(coords.positions)

    # Minimize Energy
    simulation.minimizeEnergy(tolerance=1.0*unit.kilojoules_per_mole, maxIterations=5000)

    # Save final coordinates
    positions = simulation.context.getState(getPositions=True).getPositions()
    with open(os.path.join(folder, 'minimized.pdb'), 'w') as file:
        app.PDBFile.writeFile(simulation.topology, positions, file, keepIds=True)
```

```
2020-08-19 10:30:20 PM Running minimization in window a000...
2020-08-19 10:30:21 PM Running minimization in window a001...
2020-08-19 10:30:21 PM Running minimization in window a002...
2020-08-19 10:30:21 PM Running minimization in window a003...
2020-08-19 10:30:21 PM Running minimization in window a004...
2020-08-19 10:30:21 PM Running minimization in window a005...
2020-08-19 10:30:21 PM Running minimization in window a006...
2020-08-19 10:30:21 PM Running minimization in window a007...
2020-08-19 10:30:22 PM Running minimization in window a008...
2020-08-19 10:30:22 PM Running minimization in window a009...
2020-08-19 10:30:22 PM Running minimization in window a010...
2020-08-19 10:30:22 PM Running minimization in window a011...
2020-08-19 10:30:22 PM Running minimization in window a012...
2020-08-19 10:30:22 PM Running minimization in window a013...
2020-08-19 10:30:22 PM Running minimization in window p000...
2020-08-19 10:30:23 PM Running minimization in window p001...
2020-08-19 10:30:23 PM Running minimization in window p002...
2020-08-19 10:30:23 PM Running minimization in window p003...
2020-08-19 10:30:23 PM Running minimization in window p004...
2020-08-19 10:30:23 PM Running minimization in window p005...
2020-08-19 10:30:23 PM Running minimization in window p006...
2020-08-19 10:30:24 PM Running minimization in window p007...
2020-08-19 10:30:24 PM Running minimization in window p008...
2020-08-19 10:30:24 PM Running minimization in window p009...
2020-08-19 10:30:24 PM Running minimization in window p010...
2020-08-19 10:30:24 PM Running minimization in window p011...
```

(continues on next page)

(continued from previous page)

```
2020-08-19 10:30:24 PM Running minimization in window p012...
2020-08-19 10:30:24 PM Running minimization in window p013...
2020-08-19 10:30:25 PM Running minimization in window p014...
2020-08-19 10:30:25 PM Running minimization in window p015...
2020-08-19 10:30:25 PM Running minimization in window p016...
2020-08-19 10:30:25 PM Running minimization in window p017...
```

- Here we will skip the equilibration step and go straight to production!

Production Run

[26]:
```
for window in window_list:
    folder = os.path.join(work_dir, window)
    logging.info(f"Running production in window {window}...")

    # Load XML and input coordinates
    with open(os.path.join(folder, 'system.xml'), 'r') as file:
        system = openmm.XmlSerializer.deserialize(file.read())
    coords = app.PDBFile(os.path.join(folder, 'minimized.pdb'))

    # Integrator
    integrator = openmm.LangevinIntegrator(300 * unit.kelvin, 1.0 / unit.picoseconds, 1.
↪0 * unit.femtoseconds)

    # Reporters
    dcd_reporter = app.DCDReporter(os.path.join(folder, 'production.dcd'), 500)
    state_reporter = app.StateDataReporter(
        os.path.join(folder, 'production.log'),
        500,
        step=True,
        kineticEnergy=True,
        potentialEnergy=True,
        totalEnergy=True,
        temperature=True,
    )

    # Simulation Object
    simulation = app.Simulation(coords.topology, system, integrator)
    simulation.context.setPositions(coords.positions)
    simulation.reporters.append(dcd_reporter)
    simulation.reporters.append(state_reporter)

    # MD steps
    simulation.step(50000)

    # Save final coordinates
    positions = simulation.context.getState(getPositions=True).getPositions()
    with open(os.path.join(folder, 'production.pdb'), 'w') as file:
        app.PDBFile.writeFile(simulation.topology, positions, file, keepIds=True)
```

```
2020-08-19 10:30:25 PM Running production in window a000...
```

(continues on next page)

(continued from previous page)

```
2020-08-19 10:30:28 PM Running production in window a001...
2020-08-19 10:30:31 PM Running production in window a002...
2020-08-19 10:30:33 PM Running production in window a003...
2020-08-19 10:30:36 PM Running production in window a004...
2020-08-19 10:30:39 PM Running production in window a005...
2020-08-19 10:30:42 PM Running production in window a006...
2020-08-19 10:30:45 PM Running production in window a007...
2020-08-19 10:30:47 PM Running production in window a008...
2020-08-19 10:30:50 PM Running production in window a009...
2020-08-19 10:30:53 PM Running production in window a010...
2020-08-19 10:30:56 PM Running production in window a011...
2020-08-19 10:30:59 PM Running production in window a012...
2020-08-19 10:31:01 PM Running production in window a013...
2020-08-19 10:31:04 PM Running production in window p000...
2020-08-19 10:31:07 PM Running production in window p001...
2020-08-19 10:31:10 PM Running production in window p002...
2020-08-19 10:31:13 PM Running production in window p003...
2020-08-19 10:31:15 PM Running production in window p004...
2020-08-19 10:31:18 PM Running production in window p005...
2020-08-19 10:31:21 PM Running production in window p006...
2020-08-19 10:31:24 PM Running production in window p007...
2020-08-19 10:31:26 PM Running production in window p008...
2020-08-19 10:31:29 PM Running production in window p009...
2020-08-19 10:31:32 PM Running production in window p010...
2020-08-19 10:31:35 PM Running production in window p011...
2020-08-19 10:31:38 PM Running production in window p012...
2020-08-19 10:31:40 PM Running production in window p013...
2020-08-19 10:31:43 PM Running production in window p014...
2020-08-19 10:31:46 PM Running production in window p015...
2020-08-19 10:31:49 PM Running production in window p016...
2020-08-19 10:31:52 PM Running production in window p017...
```

5.6.5 Analysis

Once the simulation is completed, we can using the `analysis` module to determine the binding free energy. We supply the location of the parameter information, a string or list for the file names (wildcards supported), the location of the windows, and the restraints on the guest.

In this example, we use the method `ti-block` which determines the free energy using **t**hermodynamic **i**integration and then estimates the standard error of the mean at each data point using blocking analysis. Bootstrapping it used to determine the uncertainty of the full thermodynamic integral for each phase.

After running `compute_free_energy()`, a dictionary called `results` will be populated, that contains the free energy and SEM for each phase of the simulation.

[27]:
```
import paprika.analysis as analysis
```

[28]:
```
free_energy = analysis.fe_calc()
free_energy.topology = "system.pdb"
free_energy.trajectory = "production.dcd"
free_energy.path = work_dir
```

(continues on next page)

(continued from previous page)

```
free_energy.restraint_list = guest_restraints
free_energy.collect_data()
free_energy.methods = ['ti-block']
free_energy.ti_matrix = "diagonal"
free_energy.boot_cycles = 1000
free_energy.compute_free_energy()
```

[29]:
```
free_energy.compute_ref_state_work([
    guest_restraints[0], guest_restraints[1], None, None,
    guest_restraints[2], None
])

binding_affinity = -1 * (
    free_energy.results["attach"]["ti-block"]["fe"] + \
    free_energy.results["pull"]["ti-block"]["fe"] + \
    free_energy.results["ref_state_work"]
)

sem = np.sqrt(
    free_energy.results["attach"]["ti-block"]["sem"]**2 + \
    free_energy.results["pull"]["ti-block"]["sem"]**2
)
```

[30]:
```
print(f"The binding affinity of butane to cucurbit[6]uril = {binding_affinity.magnitude:
↪0.2f} +/- {sem.magnitude:0.2f} kcal/mol")
```

```
The binding affinity of butane to cucurbit[6]uril = -8.96 +/- 1.14 kcal/mol
```

The value above is very close to the value obtained from running the APR calculations with Amber.

5.7 pAPRika tutorial 5 - APR/Amber with Plumed restraints

In this tutorial, we will perform APR calculations for the butane (BUT)–cucurbit[6]uril (CB6) host-guest system. This is a repeat of *Tutorial 1* using `Plumed`-based restraints and the `AMBER` MD engine. `Plumed` is a plugin for MD codes that can analyze trajectories and perform free-energy calculations on collective variables. It is a versatile plugin that can interface with a number of MD engines. Here, we will go through the process of converting APR restraints constructed with pAPRika to a `Plumed` file and run a short calculation with `sander`.

5.7.1 Initialize

Before you start

We will run the simulations in this tutorial using `sander` (*Ambertools*) and `Plumed`. Both of these should be installed in your `conda` environment if you installed *pAPRika* though the `conda` route. However, for `Plumed` to work with `sander` we first need to make sure the `PLUMED_KERNEL` environment variable is loaded (the library is called `libplumedKernel.so`). *pAPRika* should load the `Plumed` kernel automatically but let's make sure it is loaded and run the cell below.

[1]:
```
import os
'PLUMED_KERNEL' in os.environ.keys()
```

[1]:
```
False
```

If it does not exists we will load the environment in this Jupyter Notebook. Since `Plumed` is installed through `conda` the kernel will be located in your conda environment library folder. If you are running this on a Mac replace the kernel library in the cell below to `libplumedKernel.dylib`. If you compiled `Plumed` yourself and then you will need to change the path below.

[2]:
```
os.environ['PLUMED_KERNEL'] = f"{os.environ['CONDA_PREFIX']}/lib/libplumedKernel.so"
```

For running `Plumed` with Amber outside of this notebook it might be better to export the `PLUMED_KERNEL` variable into your `.bashrc` file.

Note: we can run `Plumed` with `AMBER` versions 18 and 20, but version 18 requires you to patch the source code first and recompile. Older versions of Amber are not supported. See the `Plumed` documentation for more details https://www.plumed.org/doc-v2.6/user-doc/html/index.html.

Since we have a prepared the host-guest-dummy setup from the first tutorial, we will skip the initial tleap steps and go right into initializing the restraints.

Define names

We will store the files created in this tutorial in a folder called `plumed` so we don't mix files with the previous tutorial.

[3]:
```
base_name = "cb6-but-dum"
work_dir = "plumed"
complex_dir = "complex"
```

5.7.2 Configure APR Restraints

Define anchor atoms

See *tutorial 1* for the choice of selection

[4]:
```
# Guest atoms
G1 = ":BUT@C"
G2 = ":BUT@C3"

# Host atoms
H1 = ":CB6@C"
H2 = ":CB6@C31"
H3 = ":CB6@C18"

# Dummy atoms
D1 = ":DM1"
D2 = ":DM2"
D3 = ":DM3"
```

Determine the number of windows

Before we add the restraints, it is helpful to set the λ fractions that control the strength of the force constants during attach and release, and to define the distances for the pulling phase.

The attach fractions go from 0 to 1 and we place more points at the bottom of the range to sample the curvature of $dU/d\lambda$. Next, we generally apply a distance restraint until the guest is ~18 Angstroms away from the host, in increments of 0.4 Angstroms. This distance should be at least twice the Lennard-Jones cutoff in the system. These values have worked well for us, but this is one aspect that should be carefully checked for new systems.

[5]:
```
import numpy as np
```

[6]:
```
attach_string = "0.00 0.40 0.80 1.60 2.40 4.00 5.50 8.65 11.80 18.10 24.40 37.00 49.60␣
↪74.80 100.00"
attach_fractions = [float(i) / 100 for i in attach_string.split()]

initial_distance = 6.0
pull_distances = np.arange(0.0 + initial_distance, 18.0 + initial_distance, 1.0)

release_fractions = []

windows = [len(attach_fractions), len(pull_distances), len(release_fractions)]
print(f"There are {windows} windows in this attach-pull-release calculation.")
```

```
There are [15, 18, 0] windows in this attach-pull-release calculation.
```

Load structure

[7]:
```
import parmed as pmd
```

- Load complex structure

[8]:
```
structure = pmd.load_file(
    os.path.join(complex_dir, f"{base_name}.prmtop"),
    os.path.join(complex_dir, f"{base_name}.rst7"),
    structure = True,
)
```

Host Static Restraints

See tutorial 1 for an explanation of the static restraints

[9]:
```
import paprika.restraints as restraints
```

[10]:
```
static_restraints = []
```

[11]:
```
r = restraints.static_DAT_restraint(restraint_mask_list = [D1, H1],
                                    num_window_list = windows,
                                    ref_structure = structure,
                                    force_constant = 5.0,
                                    amber_index=True)
```

(continues on next page)

(continued from previous page)

```
static_restraints.append(r)
```

[12]:
```
r = restraints.static_DAT_restraint(restraint_mask_list = [D2, D1, H1],
                                    num_window_list = windows,
                                    ref_structure = structure,
                                    force_constant = 100.0,
                                    amber_index=True)

static_restraints.append(r)
```

[13]:
```
r = restraints.static_DAT_restraint(restraint_mask_list = [D3, D2, D1, H1],
                                    num_window_list = windows,
                                    ref_structure = structure,
                                    force_constant = 100.0,
                                    amber_index=True)

static_restraints.append(r)
```

[14]:
```
r = restraints.static_DAT_restraint(restraint_mask_list = [D1, H1, H2],
                                    num_window_list = windows,
                                    ref_structure = structure,
                                    force_constant = 100.0,
                                    amber_index=True)

static_restraints.append(r)
```

[15]:
```
r = restraints.static_DAT_restraint(restraint_mask_list = [D2, D1, H1, H2],
                                    num_window_list = windows,
                                    ref_structure = structure,
                                    force_constant = 100.0,
                                    amber_index=True)

static_restraints.append(r)
```

[16]:
```
r = restraints.static_DAT_restraint(restraint_mask_list = [D1, H1, H2, H3],
                                    num_window_list = windows,
                                    ref_structure = structure,
                                    force_constant = 100.0,
                                    amber_index=True)

static_restraints.append(r)
```

Guest translational and rotational restraints

See tutorial 1 for an explanation of the guest restraints

[17]:
```
guest_restraints = []
```

[18]:
```
r = restraints.DAT_restraint()
r.mask1 = D1
r.mask2 = G1
r.topology = structure
r.auto_apr = True
r.continuous_apr = True
r.amber_index = True

r.attach["target"] = pull_distances[0]          # Angstroms
r.attach["fraction_list"] = attach_fractions
r.attach["fc_final"] = 5.0                      # kcal/mol/Angstroms**2

r.pull["target_final"] = 24.0                   # Angstroms
r.pull["num_windows"] = windows[1]

r.initialize()
guest_restraints.append(r)
```

[19]:
```
r = restraints.DAT_restraint()
r.mask1 = D2
r.mask2 = D1
r.mask3 = G1
r.topology = structure
r.auto_apr = True
r.continuous_apr = True
r.amber_index = True

r.attach["target"] = 180.0                      # Degrees
r.attach["fraction_list"] = attach_fractions
r.attach["fc_final"] = 100.0                    # kcal/mol/radian**2

r.pull["target_final"] = 180.0                  # Degrees
r.pull["num_windows"] = windows[1]

r.initialize()
guest_restraints.append(r)
```

[20]:
```
r = restraints.DAT_restraint()
r.mask1 = D1
r.mask2 = G1
r.mask3 = G2
r.topology = structure
r.auto_apr = True
r.continuous_apr = True
r.amber_index = True

r.attach["target"] = 180.0                      # Degrees
```

(continues on next page)

(continued from previous page)

```
r.attach["fraction_list"] = attach_fractions
r.attach["fc_final"] = 100.0                    # kcal/mol/radian**2

r.pull["target_final"] = 180.0                  # Degrees
r.pull["num_windows"] = windows[1]

r.initialize()
guest_restraints.append(r)
```

Create APR windows

We use the guest restraints to create a list of windows with the appropriate names and then create the directories.

[21]:
```
from paprika.restraints.utils import create_window_list
```

[22]:
```
window_list = create_window_list(guest_restraints)
```

[23]:
```
if not os.path.isdir(work_dir):
    os.makedirs(work_dir)

for window in window_list:
    folder = os.path.join(work_dir, window)
    if not os.path.isdir(folder):
        os.makedirs(os.path.join(work_dir, window))
```

Write APR restraints to Plumed format

In this section we create an instance of `Plumed()` from `paprika.restraints.plumed`, which is a class to generate the `Plumed` restraint files. We need to specify the list of restraints used throughout the APR calculations and the corresponding windows list. In this tutorial we will print the **host static** restraints and the **guest** restraints. The `Plumed` class includes a method to add restraints to dummy atoms (`add_dummy_atoms_to_file`) but we will not do that here. Instead we will use the built-in position restraints feature in Amber (see *Simulation* section below).

Note: be careful when specifiying the force constants in `DAT_restraints`. We follow the Amber (and CHARMM) convention where the force constant is already multiplied by a factor of 1/2 but `Plumed` requires the user to specify the force constant without this factor, i.e.

$$U_{amber} = K_{amber}(r - r_0)^2 \tag{5.1}$$

$$U_{plumed} = \frac{1}{2}k_{plumed}(r - r_0)^2 \tag{5.2}$$

thus $k_{plumed} = 2 \times K_{amber}$. If Amber force constants was used in generating the `DAT_restraints` (the case in this tutorial) we need to set the variable `uses_legacy_k` to `True` (this is on by default).

[24]:
```
from paprika.restraints.plumed import Plumed
```

[25]:
```
restraints_list = (static_restraints + guest_restraints)

plumed = Plumed()
plumed.file_name = 'plumed.dat'
```

(continues on next page)

(continued from previous page)

```
plumed.path = work_dir
plumed.window_list = window_list
plumed.restraint_list = restraints_list
plumed.uses_legacy_k = True

plumed.dump_to_file()
```

5.7.3 Prepare host-guest system

Translate guest molecule

For the attach windows, we will use the initial, bound coordinates for the host-guest complex. Only the force constants change during this phase, so a single set of coordinates is sufficient. For the pull windows, we will translate the guest to the target value of the restraint before solvation, and for the release windows, we will use the coordinates from the final pull window.

[26]:
```
import shutil
```

[27]:
```
for window in window_list:
    if window[0] == "a":
        shutil.copy(os.path.join(complex_dir, f"{base_name}.prmtop"),
                    os.path.join(work_dir, window, f"{base_name}.prmtop"))
        shutil.copy(os.path.join(complex_dir, f"{base_name}.rst7"),
                    os.path.join(work_dir, window, f"{base_name}.rst7"))

    elif window[0] == "p":
        structure = pmd.load_file(
            os.path.join(complex_dir, f"{base_name}.prmtop"),
            os.path.join(complex_dir, f"{base_name}.rst7"),
            structure = True
        )
        target_difference = guest_restraints[0].phase['pull']['targets'][int(window[1:
↪])] -\
                            guest_restraints[0].pull['target_initial']
        print(f"In window {window} we will translate the guest {target_difference.
↪magnitude:0.1f}.")

        for atom in structure.atoms:
            if atom.residue.name == "BUT":
                atom.xz += target_difference.magnitude

        structure.save(os.path.join(work_dir, window, f"{base_name}.prmtop"),␣
↪overwrite=True)
        structure.save(os.path.join(work_dir, window, f"{base_name}.rst7"),␣
↪overwrite=True)
```

```
In window p000 we will translate the guest 0.0 Angstroms.
In window p001 we will translate the guest 1.1 Angstroms.
In window p002 we will translate the guest 2.1 Angstroms.
In window p003 we will translate the guest 3.2 Angstroms.
In window p004 we will translate the guest 4.2 Angstroms.
```

(continues on next page)

(continued from previous page)

```
In window p005 we will translate the guest 5.3 Angstroms.
In window p006 we will translate the guest 6.4 Angstroms.
In window p007 we will translate the guest 7.4 Angstroms.
In window p008 we will translate the guest 8.5 Angstroms.
In window p009 we will translate the guest 9.5 Angstroms.
In window p010 we will translate the guest 10.6 Angstroms.
In window p011 we will translate the guest 11.6 Angstroms.
In window p012 we will translate the guest 12.7 Angstroms.
In window p013 we will translate the guest 13.8 Angstroms.
In window p014 we will translate the guest 14.8 Angstroms.
In window p015 we will translate the guest 15.9 Angstroms.
In window p016 we will translate the guest 16.9 Angstroms.
In window p017 we will translate the guest 18.0 Angstroms.
```

5.7.4 Simulation

Since we are going to run an implicit solvent simulation, we have everything ready to go. **pAPRika** has an `AMBER` module that can help setting default parameters for the simulation. There are some high level options that we set directly, like `simulation.path`, and then we call the function `config_gb_min()` to setup reasonable default simulation parameters for a minimization in the Generalized-Born ensemble. After that, we directly modify the simulation `cntrl` section to apply the positional restraints on the dummy atoms.

We will run the simulations with `sander` but it is also possible and faster to run this with `pmemd` or `pmemd.cuda` if you have them installed.

Note: The difference here compared to Tutorial 1 is that instead of specifying a `simulation.restraint_file` we will specify `simulation.plumed_file`.

Note: as explained at the *start* of this tutorial, make sure that the `PLUMED_KERNEL` environment variable is set otherwise the simulation will fail to run.

[28]:
```
from paprika.simulate import AMBER
```

Initialize logger

[29]:
```
import logging
from importlib import reload
reload(logging)

logger = logging.getLogger()
logging.basicConfig(
    format='%(asctime)s %(message)s',
    datefmt='%Y-%m-%d %I:%M:%S %p',
    level=logging.INFO
)
```

Energy Minimization

Run a quick minimization in every window. Note that we need to specify `simulation.cntrl["ntr"] = 1` to enable the positional restraints on the dummy atoms.

[37]:
```
for window in window_list:
    simulation = AMBER()
    simulation.executable = "sander"

    simulation.path = f"{work_dir}/{window}/"
    simulation.prefix = "minimize"

    simulation.topology = "cb6-but-dum.prmtop"
    simulation.coordinates = "cb6-but-dum.rst7"
    simulation.ref = "cb6-but-dum.rst7"
    simulation.plumed_file = "plumed.dat"

    simulation.config_gb_min()
    simulation.cntrl["ntr"] = 1
    simulation.cntrl["restraint_wt"] = 50.0
    simulation.cntrl["restraintmask"] = "'@DUM'"

    logger.info(f"Running minimization in window {window}...")
    simulation.run(overwrite=True)
```

```
2020-10-01 11:18:20 AM Running minimization in window a000...
2020-10-01 11:18:31 AM Running minimization in window a001...
2020-10-01 11:18:42 AM Running minimization in window a002...
2020-10-01 11:18:54 AM Running minimization in window a003...
2020-10-01 11:19:05 AM Running minimization in window a004...
2020-10-01 11:19:17 AM Running minimization in window a005...
2020-10-01 11:19:28 AM Running minimization in window a006...
2020-10-01 11:19:42 AM Running minimization in window a007...
2020-10-01 11:19:54 AM Running minimization in window a008...
2020-10-01 11:20:06 AM Running minimization in window a009...
2020-10-01 11:20:18 AM Running minimization in window a010...
2020-10-01 11:20:30 AM Running minimization in window a011...
2020-10-01 11:20:42 AM Running minimization in window a012...
2020-10-01 11:20:53 AM Running minimization in window a013...
2020-10-01 11:21:04 AM Running minimization in window p000...
2020-10-01 11:21:14 AM Running minimization in window p001...
2020-10-01 11:21:26 AM Running minimization in window p002...
2020-10-01 11:21:38 AM Running minimization in window p003...
2020-10-01 11:21:50 AM Running minimization in window p004...
2020-10-01 11:22:01 AM Running minimization in window p005...
2020-10-01 11:22:15 AM Running minimization in window p006...
2020-10-01 11:22:28 AM Running minimization in window p007...
2020-10-01 11:22:40 AM Running minimization in window p008...
2020-10-01 11:22:54 AM Running minimization in window p009...
2020-10-01 11:23:06 AM Running minimization in window p010...
2020-10-01 11:23:19 AM Running minimization in window p011...
2020-10-01 11:23:31 AM Running minimization in window p012...
2020-10-01 11:23:44 AM Running minimization in window p013...
2020-10-01 11:23:55 AM Running minimization in window p014...
```

(continues on next page)

(continued from previous page)

```
2020-10-01 11:24:07 AM Running minimization in window p015...
2020-10-01 11:24:19 AM Running minimization in window p016...
2020-10-01 11:24:30 AM Running minimization in window p017...
```

Production Run

Here we will skip the equilibration step and go straight to production!

[31]:
```
for window in window_list:
    simulation = AMBER()
    simulation.executable = "sander"

    simulation.path = f"{work_dir}/{window}/"
    simulation.prefix = "production"

    simulation.topology = "cb6-but-dum.prmtop"
    simulation.coordinates = "minimize.rst7"
    simulation.ref = "cb6-but-dum.rst7"
    simulation.plumed_file = "plumed.dat"

    simulation.config_gb_md()
    simulation.cntrl["ntr"] = 1
    simulation.cntrl["restraint_wt"] = 50.0
    simulation.cntrl["restraintmask"] = "'@DUM'"

    logger.info(f"Running production in window {window}...")
    simulation.run(overwrite=True)
```

```
2020-10-01 11:12:15 AM Running production in window a000...
2020-10-01 11:12:25 AM Running production in window a001...
2020-10-01 11:12:34 AM Running production in window a002...
2020-10-01 11:12:44 AM Running production in window a003...
2020-10-01 11:12:55 AM Running production in window a004...
2020-10-01 11:13:06 AM Running production in window a005...
2020-10-01 11:13:19 AM Running production in window a006...
2020-10-01 11:13:29 AM Running production in window a007...
2020-10-01 11:13:39 AM Running production in window a008...
2020-10-01 11:13:49 AM Running production in window a009...
2020-10-01 11:13:59 AM Running production in window a010...
2020-10-01 11:14:10 AM Running production in window a011...
2020-10-01 11:14:19 AM Running production in window a012...
2020-10-01 11:14:29 AM Running production in window a013...
2020-10-01 11:14:39 AM Running production in window p000...
2020-10-01 11:14:49 AM Running production in window p001...
2020-10-01 11:14:59 AM Running production in window p002...
2020-10-01 11:15:10 AM Running production in window p003...
2020-10-01 11:15:21 AM Running production in window p004...
2020-10-01 11:15:33 AM Running production in window p005...
2020-10-01 11:15:43 AM Running production in window p006...
2020-10-01 11:15:55 AM Running production in window p007...
2020-10-01 11:16:05 AM Running production in window p008...
```

(continues on next page)

(continued from previous page)

```
2020-10-01 11:16:18 AM Running production in window p009...
2020-10-01 11:16:31 AM Running production in window p010...
2020-10-01 11:16:45 AM Running production in window p011...
2020-10-01 11:16:56 AM Running production in window p012...
2020-10-01 11:17:05 AM Running production in window p013...
2020-10-01 11:17:16 AM Running production in window p014...
2020-10-01 11:17:26 AM Running production in window p015...
2020-10-01 11:17:37 AM Running production in window p016...
2020-10-01 11:17:49 AM Running production in window p017...
```

5.7.5 Analysis

Once the simulation is completed, we can using the `analysis` module to determine the binding free energy. We supply the location of the parameter information, a string or list for the file names (wildcards supported), the location of the windows, and the restraints on the guest.

In this example, we use the method `ti-block` which determines the free energy using **t**hermodynamic **i**integration and then estimates the standard error of the mean at each data point using blocking analysis. Bootstrapping it used to determine the uncertainty of the full thermodynamic integral for each phase.

After running `compute_free_energy()`, a dictionary called `results` will be populated, that contains the free energy and SEM for each phase of the simulation.

[32]:
```
import paprika.analysis as analysis
```

[33]:
```
free_energy = analysis.fe_calc()
free_energy.topology = "cb6-but-dum.prmtop"
free_energy.trajectory = 'production*.nc'
free_energy.path = work_dir
free_energy.restraint_list = guest_restraints
free_energy.collect_data()
free_energy.methods = ['ti-block']
free_energy.ti_matrix = "full"
free_energy.boot_cycles = 1000
free_energy.compute_free_energy()
```

We also need to calculate the free-energy cost of releasing the restraints on the guest molecule.

[34]:
```
free_energy.compute_ref_state_work([
    guest_restraints[0], guest_restraints[1], None, None,
    guest_restraints[2], None
])
```

Then we add the free-energies together and combine the uncertainties to get the binding-free energy

[35]:
```
binding_affinity = -1 * (
    free_energy.results["attach"]["ti-block"]["fe"] + \
    free_energy.results["pull"]["ti-block"]["fe"] + \
    free_energy.results["ref_state_work"]
)

sem = np.sqrt(
```

(continues on next page)

(continued from previous page)

```
    free_energy.results["attach"]["ti-block"]["sem"]**2 + \
    free_energy.results["pull"]["ti-block"]["sem"]**2
)
```

[36]:
```
print(f"The binding affinity of butane to cucurbit[6]uril = {binding_affinity.magnitude:
↪0.2f} +/- {sem.magnitude:0.2f} kcal/mol")
```

```
The binding affinity of butane to cucurbit[6]uril = -7.24 +/- 6.52 kcal/mol
```

5.8 pAPRika tutorial 6 - APR/Gromacs with Plumed restraints

In this tutorial, we will perform APR calculations for the butane (BUT)–cucurbit[6]uril (CB6) host-guest system. This tutorial is similar to *tutorial 5* where `Plumed` will be used for the restraints but we will be using the `GROMACS` MD engine instead. One other difference in this tutorial is that we will be performing the simulation with periodic boundary condition (PBC) and explicit water molecules. This is because `GROMACS` does not support implicit solvent calculations.

5.8.1 Initialize

Before you start

We will run the simulations in this tutorial using `GROMACS` and `Plumed`. `GROMACS` will need to be installed before you can run the *simulation* part of this tutorial. `Plumed`, however, should be installed in your `conda` environment if you installed *pAPRika* though the `conda` route. To use `Plumed` we first need to make sure the `PLUMED_KERNEL` environment variable is loaded (the library is called `libplumedKernel.so`). *pAPRika* should load the `Plumed` kernel automatically but let's make sure it is loaded and run the cell below.

[1]:
```
import os
'PLUMED_KERNEL' in os.environ.keys()
```

[1]:
```
True
```

If it does not exists we will load the environment in this Jupyter Notebook. Since `Plumed` is installed through `conda` the kernel will be located in your conda environment library folder. If you are running this on a MacOS, replace the kernel library in the cell below to `libplumedKernel.dylib`. If you compiled `Plumed` yourself and then you will need to change the path below.

[2]:
```
os.environ['PLUMED_KERNEL'] = f"{os.environ['CONDA_PREFIX']}/lib/libplumedKernel.so"
```

For running `Plumed` outside of this notebook it might be better to export the `PLUMED_KERNEL` variable into your `.bashrc` file.

Note: we can run `Plumed` with `GROMACS` *versions 2020.2* but the user is required to patch the source code. Unlike `AMBER` and `LAMMPS`, `GROMACS` does not come with a patched code out-of-the-box and the `conda-forge` repository also do not contain a patched version of `GROMACS`. See the `Plumed` documentation for more details on how to patch and recompile the `GROMACS` code https://www.plumed.org/doc-v2.6/user-doc/html/index.html.

Note: We will be running the CB6-BUT host-guest system in explicit solvent because `GROMACS` does not support implicit solvent calculation. Thus, we recommend running the simulation on a GPU.

Since we have a prepared the host-guest-dummy setup from the first tutorial, we will skip the initial tleap steps and go right into initializing the restraints. If you haven't completed *tutorial 1* please go back and run the notebook to generate the starting structure.

Define names

We will store the files created in this tutorial in a folder called `gromacs-plumed` so we don't mix files with the previous tutorial.

[3]:
```
base_name = "cb6-but-dum"
work_dir = "gromacs-plumed"
complex_dir = "complex"
data = "../../../paprika/data/cb6-but"
```

5.8.2 Configure APR Restraints

Define anchor atoms

See *tutorial 1* for the choice of selection

[4]:
```
# Guest atoms
G1 = ":BUT@C"
G2 = ":BUT@C3"

# Host atoms
H1 = ":CB6@C"
H2 = ":CB6@C31"
H3 = ":CB6@C18"

# Dummy atoms
D1 = ":DM1"
D2 = ":DM2"
D3 = ":DM3"
```

Determine the number of windows

Before we add the restraints, it is helpful to set the λ fractions that control the strength of the force constants during attach and release, and to define the distances for the pulling phase.

The attach fractions go from 0 to 1 and we place more points at the bottom of the range to sample the curvature of $dU/d\lambda$. Next, we generally apply a distance restraint until the guest is ~18 Angstroms away from the host, in increments of 0.4 Angstroms. This distance should be at least twice the Lennard-Jones cutoff in the system. These values have worked well for us, but this is one aspect that should be carefully checked for new systems.

[5]:
```
import numpy as np
```

[6]:
```
attach_string = "0.00 0.40 0.80 1.60 2.40 4.00 5.50 8.65 11.80 18.10 24.40 37.00 49.60␣
↪74.80 100.00"
attach_fractions = [float(i) / 100 for i in attach_string.split()]

initial_distance = 6.0
pull_distances = np.arange(0.0 + initial_distance, 18.0 + initial_distance, 1.0)

release_fractions = []
```

(continues on next page)

(continued from previous page)

```
windows = [len(attach_fractions), len(pull_distances), len(release_fractions)]
print(f"There are {windows} windows in this attach-pull-release calculation.")
```

```
There are [15, 18, 0] windows in this attach-pull-release calculation.
```

Load structure

[7]:
```
import parmed as pmd
```

- Load complex structure

[8]:
```
structure = pmd.load_file(
    os.path.join(complex_dir, f"{base_name}.prmtop"),
    os.path.join(complex_dir, f"{base_name}.rst7"),
    structure = True,
)
```

Host Static Restraints

See tutorial 1 for an explanation of the static restraints.

[9]:
```
import paprika.restraints as restraints
```

[10]:
```
static_restraints = []
```

[11]:
```
r = restraints.static_DAT_restraint(restraint_mask_list = [D1, H1],
                                    num_window_list = windows,
                                    ref_structure = structure,
                                    force_constant = 5.0,
                                    amber_index=True)

static_restraints.append(r)
```

[12]:
```
r = restraints.static_DAT_restraint(restraint_mask_list = [D2, D1, H1],
                                    num_window_list = windows,
                                    ref_structure = structure,
                                    force_constant = 100.0,
                                    amber_index=True)

static_restraints.append(r)
```

[13]:
```
r = restraints.static_DAT_restraint(restraint_mask_list = [D3, D2, D1, H1],
                                    num_window_list = windows,
                                    ref_structure = structure,
                                    force_constant = 100.0,
                                    amber_index=True)

static_restraints.append(r)
```

[14]:
```
r = restraints.static_DAT_restraint(restraint_mask_list = [D1, H1, H2],
                                    num_window_list = windows,
                                    ref_structure = structure,
                                    force_constant = 100.0,
                                    amber_index=True)

static_restraints.append(r)
```

[15]:
```
r = restraints.static_DAT_restraint(restraint_mask_list = [D2, D1, H1, H2],
                                    num_window_list = windows,
                                    ref_structure = structure,
                                    force_constant = 100.0,
                                    amber_index=True)

static_restraints.append(r)
```

[16]:
```
r = restraints.static_DAT_restraint(restraint_mask_list = [D1, H1, H2, H3],
                                    num_window_list = windows,
                                    ref_structure = structure,
                                    force_constant = 100.0,
                                    amber_index=True)

static_restraints.append(r)
```

Guest translational and rotational restraints

See tutorial 1 for an explanation of the guest restraints

[17]:
```
guest_restraints = []
```

[18]:
```
r = restraints.DAT_restraint()
r.mask1 = D1
r.mask2 = G1
r.topology = structure
r.auto_apr = True
r.continuous_apr = True
r.amber_index = True

r.attach["target"] = pull_distances[0]          # Angstroms
r.attach["fraction_list"] = attach_fractions
r.attach["fc_final"] = 5.0                      # kcal/mol/Angstroms**2

r.pull["target_final"] = 24.0                   # Angstroms
r.pull["num_windows"] = windows[1]

r.initialize()
guest_restraints.append(r)
```

[19]:
```
r = restraints.DAT_restraint()
r.mask1 = D2
```

(continues on next page)

(continued from previous page)

```
r.mask2 = D1
r.mask3 = G1
r.topology = structure
r.auto_apr = True
r.continuous_apr = True
r.amber_index = True

r.attach["target"] = 180.0                      # Degrees
r.attach["fraction_list"] = attach_fractions
r.attach["fc_final"] = 100.0                    # kcal/mol/radian**2

r.pull["target_final"] = 180.0                  # Degrees
r.pull["num_windows"] = windows[1]

r.initialize()
guest_restraints.append(r)
```

[20]:
```
r = restraints.DAT_restraint()
r.mask1 = D1
r.mask2 = G1
r.mask3 = G2
r.topology = structure
r.auto_apr = True
r.continuous_apr = True
r.amber_index = True

r.attach["target"] = 180.0                      # Degrees
r.attach["fraction_list"] = attach_fractions
r.attach["fc_final"] = 100.0                    # kcal/mol/radian**2

r.pull["target_final"] = 180.0                  # Degrees
r.pull["num_windows"] = windows[1]

r.initialize()
guest_restraints.append(r)
```

Create APR windows

We use the guest restraints to create a list of windows with the appropriate names and then create the directories.

[21]:
```
from paprika.restraints.utils import create_window_list
```

[22]:
```
window_list = create_window_list(guest_restraints)
```

[23]:
```
if not os.path.isdir(work_dir):
    os.makedirs(work_dir)

for window in window_list:
    folder = os.path.join(work_dir, window)
    if not os.path.isdir(folder):
```

(continues on next page)

(continued from previous page)

```
        os.makedirs(os.path.join(work_dir, window))
```

Write APR restraints to Plumed format

In this section we create an instance of `Plumed()` from `paprika.restraints.plumed`, which is a class to generate the `Plumed` restraint files. We need to specify the list of restraints used throughout the APR calculations and the corresponding windows list. In this tutorial we will print the **host static** restraints and the **guest** restraints. The `Plumed` class includes a method to add restraints to dummy atoms (`add_dummy_atoms_to_file`). We will add the positional restraints on the dummy atoms after solvation of the host-guest complex.

Note: be careful when specifying the force constants in `DAT_restraints`. We follow the AMBER (and CHARMM) convention where the force constant is already multiplied by a factor of 1/2 but `Plumed` requires the user to specify the force constant without this factor, i.e.

$$U_{amber} = K_{amber}(r - r_0)^2 \tag{5.3}$$

$$U_{plumed} = \frac{1}{2}k_{plumed}(r - r_0)^2 \tag{5.4}$$

thus $k_{plumed} = 2 \times K_{amber}$. If AMBER force constants was used in generating the `DAT_restraints` (the case in this tutorial) we need to set the variable `uses_legacy_k` to `True` (this is on by default).

[24]:
```
from paprika.restraints.plumed import Plumed
```

[25]:
```
restraints_list = (static_restraints + guest_restraints)

plumed = Plumed()
plumed.file_name = 'plumed.dat'
plumed.path = work_dir
plumed.window_list = window_list
plumed.restraint_list = restraints_list
plumed.uses_legacy_k = True

plumed.dump_to_file()
```

5.8.3 Prepare host-guest system

Translate guest molecule

For the attach windows, we will use the initial, bound coordinates for the host-guest complex. Only the force constants change during this phase, so a single set of coordinates is sufficient. For the pull windows, we will translate the guest to the target value of the restraint before solvation, and for the release windows, we will use the coordinates from the final pull window.

[26]:
```
import shutil
```

[27]:
```
for window in window_list:
    if window[0] == "a":
        shutil.copy(os.path.join(complex_dir, f"{base_name}.prmtop"),
                    os.path.join(work_dir, window, f"{base_name}.prmtop"))
        shutil.copy(os.path.join(complex_dir, f"{base_name}.rst7"),
```

(continues on next page)

(continued from previous page)

```
                    os.path.join(work_dir, window, f"{base_name}.rst7"))

    elif window[0] == "p":
        structure = pmd.load_file(
            os.path.join(complex_dir, f"{base_name}.prmtop"),
            os.path.join(complex_dir, f"{base_name}.rst7"),
            structure = True
        )
        target_difference = guest_restraints[0].phase['pull']['targets'][int(window[1:
↪])] -\
                            guest_restraints[0].pull['target_initial']
        print(f"In window {window} we will translate the guest {target_difference.
↪magnitude:0.1f}.")

        for atom in structure.atoms:
            if atom.residue.name == "BUT":
                atom.xz += target_difference.magnitude

        structure.save(os.path.join(work_dir, window, f"{base_name}.prmtop"),␣
↪overwrite=True)
        structure.save(os.path.join(work_dir, window, f"{base_name}.rst7"),␣
↪overwrite=True)
```

```
In window p000 we will translate the guest 0.0 Angstroms.
In window p001 we will translate the guest 1.1 Angstroms.
In window p002 we will translate the guest 2.1 Angstroms.
In window p003 we will translate the guest 3.2 Angstroms.
In window p004 we will translate the guest 4.2 Angstroms.
In window p005 we will translate the guest 5.3 Angstroms.
In window p006 we will translate the guest 6.4 Angstroms.
In window p007 we will translate the guest 7.4 Angstroms.
In window p008 we will translate the guest 8.5 Angstroms.
In window p009 we will translate the guest 9.5 Angstroms.
In window p010 we will translate the guest 10.6 Angstroms.
In window p011 we will translate the guest 11.6 Angstroms.
In window p012 we will translate the guest 12.7 Angstroms.
In window p013 we will translate the guest 13.8 Angstroms.
In window p014 we will translate the guest 14.8 Angstroms.
In window p015 we will translate the guest 15.9 Angstroms.
In window p016 we will translate the guest 16.9 Angstroms.
In window p017 we will translate the guest 18.0 Angstroms.
```

Solvate the host-guest system

Here we will solvate the host-guest system in each window with 2000 water molecules. Since `Gromacs` cannot read in `Amber` topology and coordinates files we will have to convert them to the appropriate formats. `TLeap` has a built-in function called `TLeap.convert_to_gromacs()`, which will convert the `Amber` files to `Gromacs`. We also need to add positional restraints on dummy atoms to the `plumed.dat` file and this is done by invoking the function `plumed.add_dummy_atoms_to_file`

[28]:
```
from paprika.build.system import TLeap
```

[29]:
```
for window in window_list:
    print(f"Solvating system in {window}.")

    structure = pmd.load_file(
        os.path.join(work_dir, window, f"{base_name}.prmtop"),
        os.path.join(work_dir, window, f"{base_name}.rst7"),
        structure=True,
    )

    if not os.path.exists(os.path.join(work_dir, window, f"{base_name}.pdb")):
        structure.save(os.path.join(work_dir, window, f"{base_name}.pdb"))

    system = TLeap()
    system.output_path = os.path.join(work_dir, window)
    system.output_prefix = f"{base_name}-sol"

    system.target_waters = 2000
    system.neutralize = True
    system.add_ions = ["Na+", 6, "Cl-", 6]
    system.pbc_type = "rectangular"
    system.template_lines = [
        "source leaprc.gaff",
        "source leaprc.water.tip3p",
        f"loadamberparams {data}/cb6.frcmod",
        "loadamberparams ../../complex/dummy.frcmod",
        f"CB6 = loadmol2 {data}/cb6.mol2",
        f"BUT = loadmol2 {data}/but.mol2",
        "DM1 = loadmol2 ../../complex/dm1.mol2",
        "DM2 = loadmol2 ../../complex/dm2.mol2",
        "DM3 = loadmol2 ../../complex/dm3.mol2",
        "model = loadpdb cb6-but-dum.pdb",
    ]
    system.build()

    # Convert Amber to Gromacs
    system.convert_to_gromacs(overwrite=True)

    # Add dummy atom restraints - load in the Gromacs files instead
    # of the amber files because the coordinates a shifted.
    structure = pmd.load_file(
        os.path.join(work_dir, window, f"{base_name}-sol.top"),
        xyz=os.path.join(work_dir, window, f"{base_name}-sol.gro"),
        structure=True
    )
    plumed.add_dummy_atoms_to_file(structure,window=window)
```

```
Solvating system in a000.
Solvating system in a001.
Solvating system in a002.
Solvating system in a003.
Solvating system in a004.
Solvating system in a005.
Solvating system in a006.
```

(continues on next page)

(continued from previous page)

```
Solvating system in a007.
Solvating system in a008.
Solvating system in a009.
Solvating system in a010.
Solvating system in a011.
Solvating system in a012.
Solvating system in a013.
Solvating system in p000.
Solvating system in p001.
Solvating system in p002.
Solvating system in p003.
Solvating system in p004.
Solvating system in p005.
Solvating system in p006.
Solvating system in p007.
Solvating system in p008.
Solvating system in p009.
Solvating system in p010.
Solvating system in p011.
Solvating system in p012.
Solvating system in p013.
Solvating system in p014.
Solvating system in p015.
Solvating system in p016.
Solvating system in p017.
```

5.8.4 Simulation

Now that we have a solvated structure and a `plumed.dat` file with dummy atoms position restraints, we have everything ready to go. **pAPRika** has a `GROMACS` wrapper module that can help setting default parameters for the simulation. There are some high level options that we set directly, like `simulation.path`, and then we call the function `config_pbc_min()` to setup reasonable default simulation parameters for a minimization and production run. See the `pAPRika` documentation for more details.

`GROMACS`, unlike `AMBER` and `OpenMM`, have a higher dependency on CPUs, so more than one processor for every GPU is preferred to deliver better speed. Setting the number of processors requires using different keyword depending on how `GROMACS` was compiled. The cells below were run using `GROMACS` that is compiled with MPI and hence the suffix, i.e., `gmx_mpi` (for a non-MPI version, the executable is usually called `gmx`). We set the number of processors with `simulation.n_threads = 8` and the GPU device with `simulation.gpu_devices = 0`.

Note: For host-guest systems I find that the non-MPI version is considerably much faster. Also, 12 CPUs with a single GPU seems to be a good combination with a non-MPI version.

By default, `pAPRika` will use `-ntomp` to specify the number of CPUs when executing `mdrun` if the executable is `gmx_mpi` and `-nt` for `gmx`. If these default settings do not work with the `GROMACS` currently installed then we can use `simulation.custom_mdrun_command` to override the default, e.g.,

```
simulation.custom_mdrun_command = "-ntmpi 8 -gpu_id 1"
```

or

```
simulation.custom_mdrun_command = "mpirun -np 4 gmx_mpi mdrun -ntomp 8 -gpu_id 4"
```

Note: We do not need to run `grompp` beforehand because the simulation wrapper in `pAPRika` takes care of this automatically when we call the `simulation.run()` method. We also set `simulation.grompp_maxwarn = 5` to ignore warnings about the total charge of the system (the total charge is slightly off from neutral).

Note: as explained at the *start* of this tutorial, make sure that the `PLUMED_KERNEL` environment variable is set otherwise the simulation will fail to run.

[30]:
```
from paprika.simulate import GROMACS
```

Initialize logger

[31]:
```
import logging
from importlib import reload
reload(logging)

logger = logging.getLogger()
logging.basicConfig(
    format='%(asctime)s %(message)s',
    datefmt='%Y-%m-%d %I:%M:%S %p',
    level=logging.INFO
)
```

Energy Minimization

[32]:
```
for window in window_list:
    simulation = GROMACS()
    simulation.executable = "gmx_mpi"
    simulation.n_threads = 8
    simulation.gpu_devices = 0
#     simulation.custom_mdrun_command = "-ntomp 8 -gpu_id 6"
    simulation.path = os.path.join(work_dir, window)
    simulation.maxwarn = 5

    simulation.prefix = "minimize"
    simulation.topology = f"{base_name}-sol.top"
    simulation.coordinates = f"{base_name}-sol.gro"
    simulation.plumed_file = "plumed.dat"

    simulation.config_pbc_min()

    logging.info(f"Running minimization in window {window}...")
    simulation.run(overwrite=True)
```

```
2020-09-25 05:22:17 PM Running minimization in window a000...
2020-09-25 05:22:36 PM Running minimization in window a001...
2020-09-25 05:22:56 PM Running minimization in window a002...
2020-09-25 05:23:15 PM Running minimization in window a003...
2020-09-25 05:23:37 PM Running minimization in window a004...
2020-09-25 05:23:56 PM Running minimization in window a005...
2020-09-25 05:24:16 PM Running minimization in window a006...
2020-09-25 05:24:33 PM Running minimization in window a007...
2020-09-25 05:24:52 PM Running minimization in window a008...
2020-09-25 05:25:12 PM Running minimization in window a009...
```

(continues on next page)

(continued from previous page)

```
2020-09-25 05:25:32 PM Running minimization in window a010...
2020-09-25 05:25:50 PM Running minimization in window a011...
2020-09-25 05:26:10 PM Running minimization in window a012...
2020-09-25 05:26:29 PM Running minimization in window a013...
2020-09-25 05:26:49 PM Running minimization in window p000...
2020-09-25 05:27:08 PM Running minimization in window p001...
2020-09-25 05:27:28 PM Running minimization in window p002...
2020-09-25 05:27:46 PM Running minimization in window p003...
2020-09-25 05:28:05 PM Running minimization in window p004...
2020-09-25 05:28:24 PM Running minimization in window p005...
2020-09-25 05:28:46 PM Running minimization in window p006...
2020-09-25 05:29:04 PM Running minimization in window p007...
2020-09-25 05:29:23 PM Running minimization in window p008...
2020-09-25 05:29:41 PM Running minimization in window p009...
2020-09-25 05:30:01 PM Running minimization in window p010...
2020-09-25 05:30:20 PM Running minimization in window p011...
2020-09-25 05:30:39 PM Running minimization in window p012...
2020-09-25 05:30:58 PM Running minimization in window p013...
2020-09-25 05:31:18 PM Running minimization in window p014...
2020-09-25 05:31:37 PM Running minimization in window p015...
2020-09-25 05:31:56 PM Running minimization in window p016...
2020-09-25 05:32:15 PM Running minimization in window p017...
```

NVT - Equilibration

Here, we will perform 50,000 steps of equilibration with the NVT ensemble.

[34]:
```
for window in window_list:
    simulation = GROMACS()
    simulation.executable = "gmx_mpi"
    simulation.n_threads = 8
    simulation.gpu_devices = 0
#     simulation.custom_mdrun_command = "-ntomp 8 -gpu_id 6"
    simulation.path = os.path.join(work_dir, window)
    simulation.maxwarn = 5

    simulation.prefix = "equilibration"
    simulation.topology = f"{base_name}-sol.top"
    simulation.coordinates = "minimize.gro"
    simulation.plumed_file = "plumed.dat"

    simulation.config_pbc_md(ensemble='nvt')
    simulation.control["nsteps"] = 50000

    logging.info(f"Running equilibration in window {window}...")
    simulation.run(overwrite=True)
```

```
2020-09-25 05:41:42 PM Running equilibration in window a000...
2020-09-25 05:42:26 PM Running equilibration in window a001...
2020-09-25 05:43:07 PM Running equilibration in window a002...
2020-09-25 05:43:49 PM Running equilibration in window a003...
```

(continues on next page)

(continued from previous page)

```
2020-09-25 05:44:33 PM Running equilibration in window a004...
2020-09-25 05:45:17 PM Running equilibration in window a005...
2020-09-25 05:46:01 PM Running equilibration in window a006...
2020-09-25 05:46:36 PM Running equilibration in window a007...
2020-09-25 05:47:13 PM Running equilibration in window a008...
2020-09-25 05:47:54 PM Running equilibration in window a009...
2020-09-25 05:48:36 PM Running equilibration in window a010...
2020-09-25 05:49:12 PM Running equilibration in window a011...
2020-09-25 05:49:56 PM Running equilibration in window a012...
2020-09-25 05:50:31 PM Running equilibration in window a013...
2020-09-25 05:51:11 PM Running equilibration in window p000...
2020-09-25 05:51:53 PM Running equilibration in window p001...
2020-09-25 05:52:36 PM Running equilibration in window p002...
2020-09-25 05:53:20 PM Running equilibration in window p003...
2020-09-25 05:54:02 PM Running equilibration in window p004...
2020-09-25 05:54:44 PM Running equilibration in window p005...
2020-09-25 05:55:26 PM Running equilibration in window p006...
2020-09-25 05:56:09 PM Running equilibration in window p007...
2020-09-25 05:56:44 PM Running equilibration in window p008...
2020-09-25 05:57:27 PM Running equilibration in window p009...
2020-09-25 05:58:05 PM Running equilibration in window p010...
2020-09-25 05:58:48 PM Running equilibration in window p011...
2020-09-25 05:59:31 PM Running equilibration in window p012...
2020-09-25 06:00:14 PM Running equilibration in window p013...
2020-09-25 06:00:59 PM Running equilibration in window p014...
2020-09-25 06:01:41 PM Running equilibration in window p015...
2020-09-25 06:02:20 PM Running equilibration in window p016...
2020-09-25 06:03:02 PM Running equilibration in window p017...
```

Production Run

We will run the production phase for 50,000 steps. **Note**: a proper production run would require simulation in the tens of nanoseconds per window.

[35]:
```
for window in window_list:
    simulation = GROMACS()
    simulation.executable = "gmx_mpi"
    simulation.n_threads = 8
    simulation.gpu_devices = 0
#     simulation.custom_mdrun_command = "-ntomp 8 -gpu_id 6"
    simulation.path =  os.path.join(work_dir, window)
    simulation.maxwarn = 5

    simulation.prefix = "production"
    simulation.topology = f"{base_name}-sol.top"
    simulation.coordinates = "equilibration.gro"
    simulation.checkpoint = "equilibration.cpt"
    simulation.plumed_file = "plumed.dat"

    simulation.config_pbc_md(ensemble='npt')
    simulation.control["nsteps"] = 50000
```

(continues on next page)

(continued from previous page)

```
    logging.info(f"Running production in window {window}...")
    simulation.run(overwrite=True)
```

```
2020-09-25 06:04:07 PM Running production in window a000...
2020-09-25 06:04:49 PM Running production in window a001...
2020-09-25 06:05:33 PM Running production in window a002...
2020-09-25 06:06:17 PM Running production in window a003...
2020-09-25 06:07:02 PM Running production in window a004...
2020-09-25 06:07:45 PM Running production in window a005...
2020-09-25 06:08:30 PM Running production in window a006...
2020-09-25 06:09:14 PM Running production in window a007...
2020-09-25 06:09:56 PM Running production in window a008...
2020-09-25 06:10:37 PM Running production in window a009...
2020-09-25 06:11:22 PM Running production in window a010...
2020-09-25 06:12:04 PM Running production in window a011...
2020-09-25 06:12:49 PM Running production in window a012...
2020-09-25 06:13:39 PM Running production in window a013...
2020-09-25 06:14:23 PM Running production in window p000...
2020-09-25 06:15:07 PM Running production in window p001...
2020-09-25 06:15:50 PM Running production in window p002...
2020-09-25 06:16:35 PM Running production in window p003...
2020-09-25 06:17:20 PM Running production in window p004...
2020-09-25 06:18:03 PM Running production in window p005...
2020-09-25 06:18:55 PM Running production in window p006...
2020-09-25 06:19:40 PM Running production in window p007...
2020-09-25 06:20:24 PM Running production in window p008...
2020-09-25 06:21:13 PM Running production in window p009...
2020-09-25 06:21:57 PM Running production in window p010...
2020-09-25 06:22:43 PM Running production in window p011...
2020-09-25 06:23:28 PM Running production in window p012...
2020-09-25 06:24:16 PM Running production in window p013...
2020-09-25 06:24:58 PM Running production in window p014...
2020-09-25 06:25:43 PM Running production in window p015...
2020-09-25 06:26:24 PM Running production in window p016...
2020-09-25 06:27:06 PM Running production in window p017...
```

5.8.5 Analysis

Once the simulation is completed, we can using the `Analysis` module to determine the binding free energy. We supply the location of the parameter information, a string or list for the file names (wildcards supported), the location of the windows, and the restraints on the guest.

In this example, we use the method `ti-block` which determines the free energy using **t**hermodynamic **i**integration and then estimates the standard error of the mean at each data point using blocking analysis. Bootstrapping it used to determine the uncertainty of the full thermodynamic integral for each phase.

After running `compute_free_energy()`, a dictionary called `results` will be populated, that contains the free energy and SEM for each phase of the simulation.

[36]:
```
import paprika.analysis as analysis
```

[37]:
```
free_energy = analysis.fe_calc()
free_energy.topology = "cb6-but-dum-sol.pdb"
free_energy.trajectory = 'production*.trr'
free_energy.path = work_dir
free_energy.restraint_list = guest_restraints
free_energy.collect_data()
free_energy.methods = ['ti-block']
free_energy.ti_matrix = "full"
free_energy.boot_cycles = 1000
free_energy.compute_free_energy()
```

We also need to calculate the free-energy cost of releasing the restraints on the guest molecule.

[38]:
```
free_energy.compute_ref_state_work([
    guest_restraints[0], guest_restraints[1], None, None,
    guest_restraints[2], None
])
```

Then we add the free-energies together and combine the uncertainties to get the binding-free energy

[39]:
```
binding_affinity = -1 * (
    free_energy.results["attach"]["ti-block"]["fe"] + \
    free_energy.results["pull"]["ti-block"]["fe"] + \
    free_energy.results["ref_state_work"]
)

sem = np.sqrt(
    free_energy.results["attach"]["ti-block"]["sem"]**2 + \
    free_energy.results["pull"]["ti-block"]["sem"]**2
)
```

[40]:
```
print(f"The binding affinity of butane to cucurbit[6]uril = {binding_affinity.magnitude:
↪0.2f} +/- {sem.magnitude:0.2f} kcal/mol")
```

```
The binding affinity of butane to cucurbit[6]uril = -7.55 +/- 1.58 kcal/mol
```

5.9 pAPRika tutorial 7 - APR/NAMD with Colvars restraints

In this tutorial, we will perform APR calculations for the butane (BUT)–cucurbit[6]uril (CB6) host-guest system with the Generalized-Born implicit solvent. This tutorial will use `NAMD` for the simulation and `Colvars` for the restraints. Although it is possible to run `NAMD` with `Plumed` we will not use `Plumed` in this tutorial. Users should refer to the previous tutorial to see how to use `Plumed` restraints.

Since we have a prepared the host-guest-dummy setup from the first tutorial, we will skip the initial tleap steps and go right into initializing the restraints. If you haven't completed *tutorial 1* please go back and run the notebook to generate the starting structure.

5.9.1 Initialize

[1]:
```
import os
```

Define names

We will store the files created in this tutorial in a folder called `namd` so we don't mix files with the previous tutorial.

[2]:
```
base_name = "cb6-but-dum"
work_dir = "namd"
complex_dir = "complex"
data = "../../../paprika/data/cb6-but"
```

5.9.2 Configure APR Restraints

Define anchor atoms

See *tutorial 1* for the choice of selection

[3]:
```
# Guest atoms
G1 = ":BUT@C"
G2 = ":BUT@C3"

# Host atoms
H1 = ":CB6@C"
H2 = ":CB6@C31"
H3 = ":CB6@C18"

# Dummy atoms
D1 = ":DM1"
D2 = ":DM2"
D3 = ":DM3"
```

Determine the number of windows

Before we add the restraints, it is helpful to set the λ fractions that control the strength of the force constants during attach and release, and to define the distances for the pulling phase.

The attach fractions go from 0 to 1 and we place more points at the bottom of the range to sample the curvature of $dU/d\lambda$. Next, we generally apply a distance restraint until the guest is ~18 Angstroms away from the host, in increments of 0.4 Angstroms. This distance should be at least twice the Lennard-Jones cutoff in the system. These values have worked well for us, but this is one aspect that should be carefully checked for new systems.

[4]:
```
import numpy as np
```

[5]:
```
attach_string = "0.00 0.40 0.80 1.60 2.40 4.00 5.50 8.65 11.80 18.10 24.40 37.00 49.60␣
↪74.80 100.00"
attach_fractions = [float(i) / 100 for i in attach_string.split()]

initial_distance = 6.0
```

(continues on next page)

(continued from previous page)

```
pull_distances = np.arange(0.0 + initial_distance, 18.0 + initial_distance, 1.0)

release_fractions = []

windows = [len(attach_fractions), len(pull_distances), len(release_fractions)]
print(f"There are {windows} windows in this attach-pull-release calculation.")
```

```
There are [15, 18, 0] windows in this attach-pull-release calculation.
```

Load structure

[6]:
```
import parmed as pmd
```

- Load complex structure

[7]:
```
structure = pmd.load_file(
    os.path.join(complex_dir, f"{base_name}.prmtop"),
    os.path.join(complex_dir, f"{base_name}.rst7"),
    structure = True,
)
```

Host Static Restraints

See tutorial 1 for an explanation of the static restraints.

[8]:
```
import paprika.restraints as restraints
```

[9]:
```
static_restraints = []
```

[10]:
```
r = restraints.static_DAT_restraint(restraint_mask_list = [D1, H1],
                                    num_window_list = windows,
                                    ref_structure = structure,
                                    force_constant = 5.0,
                                    amber_index=True)

static_restraints.append(r)
```

[11]:
```
r = restraints.static_DAT_restraint(restraint_mask_list = [D2, D1, H1],
                                    num_window_list = windows,
                                    ref_structure = structure,
                                    force_constant = 100.0,
                                    amber_index=True)

static_restraints.append(r)
```

[12]:
```
r = restraints.static_DAT_restraint(restraint_mask_list = [D3, D2, D1, H1],
                                    num_window_list = windows,
                                    ref_structure = structure,
                                    force_constant = 100.0,
```

(continues on next page)

(continued from previous page)

```
                                    amber_index=True)

static_restraints.append(r)
```

[13]:
```
r = restraints.static_DAT_restraint(restraint_mask_list = [D1, H1, H2],
                                    num_window_list = windows,
                                    ref_structure = structure,
                                    force_constant = 100.0,
                                    amber_index=True)

static_restraints.append(r)
```

[14]:
```
r = restraints.static_DAT_restraint(restraint_mask_list = [D2, D1, H1, H2],
                                    num_window_list = windows,
                                    ref_structure = structure,
                                    force_constant = 100.0,
                                    amber_index=True)

static_restraints.append(r)
```

[15]:
```
r = restraints.static_DAT_restraint(restraint_mask_list = [D1, H1, H2, H3],
                                    num_window_list = windows,
                                    ref_structure = structure,
                                    force_constant = 100.0,
                                    amber_index=True)

static_restraints.append(r)
```

Guest translational and rotational restraints

See tutorial 1 for an explanation of the guest restraints

[16]:
```
guest_restraints = []
```

[17]:
```
r = restraints.DAT_restraint()
r.mask1 = D1
r.mask2 = G1
r.topology = structure
r.auto_apr = True
r.continuous_apr = True
r.amber_index = True

r.attach["target"] = pull_distances[0]          # Angstroms
r.attach["fraction_list"] = attach_fractions
r.attach["fc_final"] = 5.0                      # kcal/mol/Angstroms**2

r.pull["target_final"] = 24.0                   # Angstroms
r.pull["num_windows"] = windows[1]
```

(continues on next page)

(continued from previous page)

```
r.initialize()
guest_restraints.append(r)
```

[18]:
```
r = restraints.DAT_restraint()
r.mask1 = D2
r.mask2 = D1
r.mask3 = G1
r.topology = structure
r.auto_apr = True
r.continuous_apr = True
r.amber_index = True

r.attach["target"] = 180.0                      # Degrees
r.attach["fraction_list"] = attach_fractions
r.attach["fc_final"] = 100.0                    # kcal/mol/radian**2

r.pull["target_final"] = 180.0                  # Degrees
r.pull["num_windows"] = windows[1]

r.initialize()
guest_restraints.append(r)
```

[19]:
```
r = restraints.DAT_restraint()
r.mask1 = D1
r.mask2 = G1
r.mask3 = G2
r.topology = structure
r.auto_apr = True
r.continuous_apr = True
r.amber_index = True

r.attach["target"] = 180.0                      # Degrees
r.attach["fraction_list"] = attach_fractions
r.attach["fc_final"] = 100.0                    # kcal/mol/radian**2

r.pull["target_final"] = 180.0                  # Degrees
r.pull["num_windows"] = windows[1]

r.initialize()
guest_restraints.append(r)
```

Create APR windows

We use the guest restraints to create a list of windows with the appropriate names and then create the directories.

[20]:
```
from paprika.restraints.utils import create_window_list
```

[21]:
```
window_list = create_window_list(guest_restraints)
```

[22]:
```
if not os.path.isdir(work_dir):
    os.makedirs(work_dir)

for window in window_list:
    folder = os.path.join(work_dir, window)
    if not os.path.isdir(folder):
        os.makedirs(os.path.join(work_dir, window))
```

Write APR restraints to a Colvars module file

In this section we create an instance of `Colvars()` from `paprika.restraints.colvars`, which is a class to generate the `Colvars` restraint files. This class is analogous to the `Plumed` class. We need to specify the list of restraints used throughout the APR calculations and the corresponding windows list. In this tutorial we will print the **host static** restraints and the **guest** restraints. The `Colvars` class includes a method to add restraints to dummy atoms (`add_dummy_atoms_to_file`). We will add the positional restraints on the dummy atoms after generating the structures for each window.

Note: be careful when specifying the force constants in `DAT_restraints`. We follow the AMBER (and CHARMM) convention where the force constant is already multiplied by a factor of 1/2 but `Colvars` (and `Plumed`) requires the user to specify the force constant without this factor, i.e.

$$U_{amber} = K_{amber}(r - r_0)^2 \tag{5.5}$$

$$U_{colvars} = \frac{1}{2}k_{colvars}(r - r_0)^2 \tag{5.6}$$

thus $k_{colvars} = 2 \times K_{amber}$. If AMBER force constants was used in generating the `DAT_restraints` (the case in this tutorial) we need to set the variable `uses_legacy_k` to `True` (this is on by default).

[23]:
```
from paprika.restraints.colvars import Colvars
```

[24]:
```
restraints_list = (static_restraints + guest_restraints)

colvars = Colvars()
colvars.file_name = "colvars.tcl"
colvars.path = work_dir
colvars.window_list = window_list
colvars.restraint_list = restraints_list
colvars_uses_legacy_k = True

colvars.dump_to_file()
```

5.9.3 Prepare host-guest system

Translate guest molecule

For the attach windows, we will use the initial, bound coordinates for the host-guest complex. Only the force constants change during this phase, so a single set of coordinates is sufficient. For the pull windows, we will translate the guest to the target value, and for the release windows, we will use the coordinates from the final pull window.

[25]:
```
import shutil
```

[26]:
```
for window in window_list:
    if window[0] == "a":
        shutil.copy(os.path.join(complex_dir, f"{base_name}.prmtop"),
                    os.path.join(work_dir, window, f"{base_name}.prmtop"))
        shutil.copy(os.path.join(complex_dir, f"{base_name}.rst7"),
                    os.path.join(work_dir, window, f"{base_name}.rst7"))

    elif window[0] == "p":
        structure = pmd.load_file(
            os.path.join(complex_dir, f"{base_name}.prmtop"),
            os.path.join(complex_dir, f"{base_name}.rst7"),
            structure = True
        )
        target_difference = guest_restraints[0].phase['pull']['targets'][int(window[1:
↪])] -\
                            guest_restraints[0].pull['target_initial']
        print(f"In window {window} we will translate the guest {target_difference.
↪magnitude:0.1f}.")

        for atom in structure.atoms:
            if atom.residue.name == "BUT":
                atom.xz += target_difference.magnitude

        structure.save(os.path.join(work_dir, window, f"{base_name}.prmtop"),␣
↪overwrite=True)
        structure.save(os.path.join(work_dir, window, f"{base_name}.rst7"),␣
↪overwrite=True)
```

```
In window p000 we will translate the guest 0.0 Angstroms.
In window p001 we will translate the guest 1.1 Angstroms.
In window p002 we will translate the guest 2.1 Angstroms.
In window p003 we will translate the guest 3.2 Angstroms.
In window p004 we will translate the guest 4.2 Angstroms.
In window p005 we will translate the guest 5.3 Angstroms.
In window p006 we will translate the guest 6.4 Angstroms.
In window p007 we will translate the guest 7.4 Angstroms.
In window p008 we will translate the guest 8.5 Angstroms.
In window p009 we will translate the guest 9.5 Angstroms.
In window p010 we will translate the guest 10.6 Angstroms.
In window p011 we will translate the guest 11.6 Angstroms.
In window p012 we will translate the guest 12.7 Angstroms.
In window p013 we will translate the guest 13.8 Angstroms.
In window p014 we will translate the guest 14.8 Angstroms.
In window p015 we will translate the guest 15.9 Angstroms.
In window p016 we will translate the guest 16.9 Angstroms.
In window p017 we will translate the guest 18.0 Angstroms.
```

Add dummy atom restraints

Here we will add positional restraints on the dummy atoms. The reference positions are extracted from the structure files we just created in each window.

[27]:
```
for window in window_list:
    folder = os.path.join(work_dir, window)

    structure = pmd.load_file(
        os.path.join(folder, f"{base_name}.prmtop"),
        os.path.join(folder, f"{base_name}.rst7"),
        structure = True
    )

    colvars.add_dummy_atom_restraints(structure, window)
```

5.9.4 Simulation

Now that we have prepared the structure and `colvars.tcl` file for each window, we have everything ready to go. **pAPRika** comes with a python wrapper for `NAMD` that can help set up default parameters for the simulation. There are some high level options that we set directly, like `simulation.path`, and then we call the function `config_gb_min()` to setup reasonable default simulation parameters for a minimization and production run. The temperature of the system will be controlled with the `Langevin` thermostat, which is the default. For running simulations with a different thermostat parse the thermostat of choice using the `Thermostat` Enum class to `config_gb_md()`. For example, if we want to run our simulations with the *Lowe-Anderson* thermostat:

```
simulation = NAMD()
...
simulation.config_gb_md(NAMD.Thermostat.LoweAnderson)
simulation.run()
```

See the pAPRika documentation for more details on the different temperature control algorithm implemented.

Note: `NAMD` version 2 is similar to `GROMACS` in that more than one thread for every GPU is preferred to deliver better speed. We set the number of processors with `simulation.n_threads = 8` and the GPU device with `simulation.gpu_devices = 0`. Starting with `NAMD` version 3 (currently in alpha phase) a smaller number of `n_threads` is required per GPU.

[28]:
```
from paprika.simulate import NAMD
```

Initialize logger

[29]:
```
import logging
from importlib import reload
reload(logging)

logger = logging.getLogger()
logging.basicConfig(
    format='%(asctime)s %(message)s',
    datefmt='%Y-%m-%d %I:%M:%S %p',
    level=logging.INFO
)
```

Energy Minimization

[30]:
```
for window in window_list:
    simulation = NAMD()
    simulation.executable = "namd2"
    simulation.n_threads = 4
#     simulation.gpu_devices = 0
    simulation.path = os.path.join(work_dir, window)

    simulation.topology = f"{base_name}.prmtop"
    simulation.coordinates = f"{base_name}.rst7"
    simulation.prefix = "minimize"
    simulation.colvars_file = "colvars.tcl"

    simulation.config_gb_min()

    logging.info(f"Running minimization in window {window}...")
    simulation.run(overwrite=True)
```

```
2020-10-09 11:31:02 AM Running minimization in window a000...
2020-10-09 11:31:07 AM Running minimization in window a001...
2020-10-09 11:31:13 AM Running minimization in window a002...
2020-10-09 11:31:20 AM Running minimization in window a003...
2020-10-09 11:31:25 AM Running minimization in window a004...
2020-10-09 11:31:30 AM Running minimization in window a005...
2020-10-09 11:31:34 AM Running minimization in window a006...
2020-10-09 11:31:39 AM Running minimization in window a007...
2020-10-09 11:31:43 AM Running minimization in window a008...
2020-10-09 11:31:48 AM Running minimization in window a009...
2020-10-09 11:31:53 AM Running minimization in window a010...
2020-10-09 11:31:59 AM Running minimization in window a011...
2020-10-09 11:32:03 AM Running minimization in window a012...
2020-10-09 11:32:08 AM Running minimization in window a013...
2020-10-09 11:32:12 AM Running minimization in window p000...
2020-10-09 11:32:16 AM Running minimization in window p001...
2020-10-09 11:32:21 AM Running minimization in window p002...
2020-10-09 11:32:25 AM Running minimization in window p003...
2020-10-09 11:32:30 AM Running minimization in window p004...
2020-10-09 11:32:34 AM Running minimization in window p005...
2020-10-09 11:32:39 AM Running minimization in window p006...
2020-10-09 11:32:43 AM Running minimization in window p007...
2020-10-09 11:32:48 AM Running minimization in window p008...
2020-10-09 11:32:56 AM Running minimization in window p009...
2020-10-09 11:33:01 AM Running minimization in window p010...
2020-10-09 11:33:05 AM Running minimization in window p011...
2020-10-09 11:33:10 AM Running minimization in window p012...
2020-10-09 11:33:14 AM Running minimization in window p013...
2020-10-09 11:33:19 AM Running minimization in window p014...
2020-10-09 11:33:23 AM Running minimization in window p015...
2020-10-09 11:33:28 AM Running minimization in window p016...
2020-10-09 11:33:32 AM Running minimization in window p017...
```

Thermalization

Here, we will do a slow heating process to demonstrate the `custom_run_command` feature of the `NAMD` wrapper. We will heat the system from 50 K to 300 K in steps of 50 K and will do a total of 500 MD steps for each temperature.

[31]:
```
for window in window_list:
    simulation = NAMD()
    simulation.executable = "namd2"
    simulation.n_threads = 4
#     simulation.gpu_devices = 0
    simulation.path = os.path.join(work_dir, window)

    simulation.topology = f"{base_name}.prmtop"
    simulation.coordinates = f"{base_name}.rst7"
    simulation.prefix = "thermalization"
    simulation.checkpoint = "minimize"
    simulation.colvars_file = "colvars.tcl"

    simulation.config_gb_md()

    # Slow heating
    simulation.custom_run_commands = [
        "for {set temp 50} {$temp <= 300} {incr temp 50} {",
        "langevinTemp $temp",
        "reinitvels $temp",
        "run 500",
        "}",
    ]

    logging.info(f"Running thermalization in window {window}...")
    simulation.run(overwrite=True)
```

```
2020-10-09 11:33:37 AM Running thermalization in window a000...
2020-10-09 11:33:39 AM Running thermalization in window a001...
2020-10-09 11:33:40 AM Running thermalization in window a002...
2020-10-09 11:33:42 AM Running thermalization in window a003...
2020-10-09 11:33:44 AM Running thermalization in window a004...
2020-10-09 11:33:46 AM Running thermalization in window a005...
2020-10-09 11:33:47 AM Running thermalization in window a006...
2020-10-09 11:33:49 AM Running thermalization in window a007...
2020-10-09 11:33:51 AM Running thermalization in window a008...
2020-10-09 11:33:53 AM Running thermalization in window a009...
2020-10-09 11:33:54 AM Running thermalization in window a010...
2020-10-09 11:33:56 AM Running thermalization in window a011...
2020-10-09 11:33:58 AM Running thermalization in window a012...
2020-10-09 11:34:00 AM Running thermalization in window a013...
2020-10-09 11:34:02 AM Running thermalization in window p000...
2020-10-09 11:34:06 AM Running thermalization in window p001...
2020-10-09 11:34:09 AM Running thermalization in window p002...
2020-10-09 11:34:13 AM Running thermalization in window p003...
2020-10-09 11:34:15 AM Running thermalization in window p004...
2020-10-09 11:34:17 AM Running thermalization in window p005...
2020-10-09 11:34:18 AM Running thermalization in window p006...
2020-10-09 11:34:20 AM Running thermalization in window p007...
```

(continues on next page)

(continued from previous page)

```
2020-10-09 11:34:22 AM Running thermalization in window p008...
2020-10-09 11:34:24 AM Running thermalization in window p009...
2020-10-09 11:34:25 AM Running thermalization in window p010...
2020-10-09 11:34:27 AM Running thermalization in window p011...
2020-10-09 11:34:29 AM Running thermalization in window p012...
2020-10-09 11:34:30 AM Running thermalization in window p013...
2020-10-09 11:34:32 AM Running thermalization in window p014...
2020-10-09 11:34:34 AM Running thermalization in window p015...
2020-10-09 11:34:36 AM Running thermalization in window p016...
2020-10-09 11:34:37 AM Running thermalization in window p017...
```

We will skip the equilibration step and go straight to production

Production Run

We will run the production phase for 5,000 steps.

Note: a proper production run would require much longer simulation.

[32]:
```
for window in window_list:
    simulation = NAMD()
    simulation.executable = "namd2"
    simulation.n_threads = 4
#     simulation.gpu_devices = 0
    simulation.path = os.path.join(work_dir, window)

    simulation.topology = f"{base_name}.prmtop"
    simulation.coordinates = f"{base_name}.rst7"
    simulation.prefix = "production"
    simulation.checkpoint = "thermalization"
    simulation.colvars_file = "colvars.tcl"

    simulation.config_gb_md()
    simulation.control["run"] = 5000

    logging.info(f"Running production in window {window}...")
    simulation.run(overwrite=True)
```

```
2020-10-09 11:34:39 AM Running production in window a000...
2020-10-09 11:34:41 AM Running production in window a001...
2020-10-09 11:34:44 AM Running production in window a002...
2020-10-09 11:34:46 AM Running production in window a003...
2020-10-09 11:34:49 AM Running production in window a004...
2020-10-09 11:34:52 AM Running production in window a005...
2020-10-09 11:34:54 AM Running production in window a006...
2020-10-09 11:34:57 AM Running production in window a007...
2020-10-09 11:35:00 AM Running production in window a008...
2020-10-09 11:35:02 AM Running production in window a009...
2020-10-09 11:35:05 AM Running production in window a010...
2020-10-09 11:35:10 AM Running production in window a011...
2020-10-09 11:35:15 AM Running production in window a012...
2020-10-09 11:35:17 AM Running production in window a013...
```

(continues on next page)

(continued from previous page)

```
2020-10-09 11:35:21 AM Running production in window p000...
2020-10-09 11:35:25 AM Running production in window p001...
2020-10-09 11:35:29 AM Running production in window p002...
2020-10-09 11:35:33 AM Running production in window p003...
2020-10-09 11:35:36 AM Running production in window p004...
2020-10-09 11:35:40 AM Running production in window p005...
2020-10-09 11:35:42 AM Running production in window p006...
2020-10-09 11:35:46 AM Running production in window p007...
2020-10-09 11:35:51 AM Running production in window p008...
2020-10-09 11:35:54 AM Running production in window p009...
2020-10-09 11:35:58 AM Running production in window p010...
2020-10-09 11:36:06 AM Running production in window p011...
2020-10-09 11:36:15 AM Running production in window p012...
2020-10-09 11:36:21 AM Running production in window p013...
2020-10-09 11:36:27 AM Running production in window p014...
2020-10-09 11:36:32 AM Running production in window p015...
2020-10-09 11:36:37 AM Running production in window p016...
2020-10-09 11:36:41 AM Running production in window p017...
```

5.9.5 Analysis

Once the simulation is completed, we can using the `Analysis` module to determine the binding free energy. We supply the location of the parameter information, a string or list for the file names (wildcards supported), the location of the windows, and the restraints on the guest.

In this example, we use the method `ti-block` which determines the free energy using **t**hermodynamic **i**integration and then estimates the standard error of the mean at each data point using blocking analysis. Bootstrapping it used to determine the uncertainty of the full thermodynamic integral for each phase.

After running `compute_free_energy()`, a dictionary called `results` will be populated, that contains the free energy and SEM for each phase of the simulation.

[33]:
```
import paprika.analysis as analysis
```

[35]:
```
free_energy = analysis.fe_calc()
free_energy.topology = "cb6-but-dum.prmtop"
free_energy.trajectory = 'production*.dcd'
free_energy.path = work_dir
free_energy.restraint_list = guest_restraints
free_energy.collect_data()
free_energy.methods = ['ti-block']
free_energy.ti_matrix = "full"
free_energy.boot_cycles = 1000
free_energy.compute_free_energy()
```

We also need to calculate the free-energy cost of releasing the restraints on the guest molecule.

[36]:
```
free_energy.compute_ref_state_work([
    guest_restraints[0], guest_restraints[1], None,
    None, guest_restraints[2], None
])
```

Then we add the free-energies together and combine the uncertainties to get the binding-free energy

[37]:
```
binding_affinity = -1 * (
    free_energy.results["attach"]["ti-block"]["fe"] + \
    free_energy.results["pull"]["ti-block"]["fe"] + \
    free_energy.results["ref_state_work"]
)

sem = np.sqrt(
    free_energy.results["attach"]["ti-block"]["sem"]**2 + \
    free_energy.results["pull"]["ti-block"]["sem"]**2
)
```

[38]:
```
print(f"The binding affinity of butane to cucurbit[6]uril = {binding_affinity.magnitude:
↪0.2f} +/- {sem.magnitude:0.2f} kcal/mol")
```

```
The binding affinity of butane to cucurbit[6]uril = -6.16 +/- 4.40 kcal/mol
```

5.10 Compiling the Docs

The docs for this project are built with [Sphinx](http://www.sphinx-doc.org/en/master/). To compile the docs, first ensure that Sphinx, and the ReadTheDocs theme are installed.:

```
conda install sphinx nbsphinx sphinx_rtd_theme
```

Once installed, you can use the `Makefile` in this directory to compile static HTML pages by:

```
make html
```

The compiled docs will be in the `_build` directory and can be viewed by opening `index.html` (which may itself be inside a directory called `html/` depending on what version of Sphinx is installed).

5.11 API

This page provides links to the documentation for classes and modules available in *paprika*.

- *Align API*
- *Analysis API*
- *Dummy Atoms API*
- *Evaluator API*
- *Restraints API*
- *Simulation Wrappers API*
- *System API*
- *Utils API*

5.11.1 Align API

The *Align* module provide a number of functions to translate and orient molecules that is useful during structure preparation.

5.11.2 Analysis API

5.11.3 Dummy Atoms API

5.11.4 Evaluator API

Analysis

Setup

GAFF

Utils

5.11.5 Restraints API

DAT Restraints

AMBER Restraints

OpenMM Restraints

Plumed Module

Colvars Module

Utility Modules

5.11.6 Simulation Wrappers API

Base Simulation Class

AMBER Simulation Wrapper

GROMACS Simulation Wrapper

NAMD Simulation Wrapper

5.11.7 System API

TLeap Wrapper

Utils

5.11.8 Utils API

5.12 Release History

5.12.1 v1.2.0

(10/26/2022)

This release adds minor new features bug fixes to the paprika code.

- Integrate OpenFF-units (a wrapper for Pint units) used in defining restraints.
- Enhances the *restraints/openmm* module to include centroid-based anchor atom selection.
- Updates the analysis module for pymbar v4.0
- Minor bug fixes

5.12.2 v1.1.0

(02/17/2021)

This release provides a number of enhancements to the paprika code and changes to the API.

- Adds support for writing APR restraints to `Plumed` and `Colvars` format.
- Refactoring of the `AMBER` simulation wrapper code and expanded to include wrappers for `GROMACS` and `NAMD`.
- Namespace refactoring for the `align`, `dummy`, and `tleap` modules (now under `paprika.build`).
- Namespace refactoring for the OpenFF-Evaluator related modules (now under `paprika.evaluator`).
- Enhancements to the `tleap` class, which includes a wrapper around `InterMol` for converting `AMBER` files to other MD formats and support for solvating with the `Bind3P` water model.
- Migrate continuous integration (CI) to Github Actions (GHA).
- Minor bug fixes.

5.12.3 v1.0.4

(08/18/2020)

- Refactor codes for `setup.py` and `analysis.py` modules to allow a smoother integration with OpenFF-Evaluator
- OpenMM restraints code is now decoupled from `setup.py` and placed in a separate file `restraints/openmm.py`
- Updated install requirements to Ambertools v20
- Added hydrogen mass repartitioning option to `TLeap` API module
- Minor bug fixes

5.12.4 v1.0.3

(12/04/2019)

Fixed API compatibility with the new pymbar v3.0.5 and the I/O for `ref_structure` in the `static_DAT_restraint` function

5.12.5 v1.0.2

(11/24/2019)

Fix parsing of `CUDA_VISIBLE_DEVICES` class property. This bugfix was intended to go into the `v1.0.1` release but was accidentally excluded.

5.12.6 v1.0.1

(11/24/2019)

Provides a more robust method to supply a random number seed to the `compute_free_energy` method.

5.12.7 v1.0.0

(11/04/2019)

- The API for `paprika.setup` and `paprika.analyze` are now converged.
- OpenMM support is proved by PropertyEstimator.
- The documentation has (mostly) been updated and expanded.

5.12.8 v0.1.1

(07/30/2019)

Minor code cleanup and fix reference values for test cases that were failing in the v0.1.0 release.

5.12.9 v0.1.0

(07/27/2019)

This release enables `paprika` to work with `propertyestimator` of the Open Force Field Toolkit, and `taproom` to setup host-guest calculations with either GAFF or SMIRNOFF-based force fields using AMBER or OpenMM as simulation backends.

5.12.10 v0.0.4

(02/19/2019)

Updates after merging Cookiecutter for Computational Molecular Sciences (CMS) Python Packages and updating tests to pass.

5.12.11 v0.0.3

(05/16/2018)

Initial code and was used to run SAMPLing.