biobb$_{w}f_{a}mber_{m}d_{s}etupDocumentation$

Release 1.0.0

Bioexcel Project

Oct 11, 2021

CONTENTS

CHAPTER

ONE

CONTENTS

1.1 AMBER Protein MD Setup tutorials using BioExcel Building Blocks (biobb)

Based on the official GROMACS tutorial.

This tutorials aim to illustrate the process of **setting up a simulation** system containing a **protein**, step by step, using the **BioExcel Building Blocks library (biobb)** wrapping the **Ambertools MD package**.

1.1.1 Settings

Biobb modules used

- biobb_io: Tools to fetch biomolecular data from public databases.
- biobb_amber: Tools to setup and run Molecular Dynamics simulations using the Ambertools MD package.
- biobb_analysis: Tools to analyse Molecular Dynamics trajectories.
- biobb_structure_utils: Tools to modify or extract information from a PDB structure file.
- biobb_chemistry: Tools to to perform chemical conversions.

Auxiliar libraries used

- nb_conda_kernels: Enables a Jupyter Notebook or JupyterLab application in one conda environment to access kernels for Python, R, and other languages found in other environments.
- nglview: Jupyter/IPython widget to interactively view molecular structures and trajectories in notebooks.
- ipywidgets: Interactive HTML widgets for Jupyter notebooks and the IPython kernel.
- plotly: Python interactive graphing library integrated in Jupyter notebooks.
- simpletraj: Lightweight coordinate-only trajectory reader based on code from GROMACS, MDAnalysis and VMD.

Conda Installation

```
git clone https://github.com/bioexcel/biobb_wf_amber_md_setup.git
cd biobb_wf_amber_md_setup
conda env create -f conda_env/environment.yml
conda activate biobb_AMBER_MDsetup_tutorials
./conda_env/post-link.sh
```

1.1.2 Launch

Protein MD Setup tutorial

```
jupyter-notebook biobb_wf_amber_md_setup/notebooks/mdsetup/biobb_amber_setup_notebook.
↪ipynb
```

Protein-Ligand Complex MD Setup tutorial

```
jupyter-notebook biobb_wf_amber_md_setup/notebooks/mdsetup_lig/biobb_amber_complex_setup_
↪notebook.ipynb
```

Constant pH MD Setup tutorial

```
jupyter-notebook biobb_wf_amber_md_setup/notebooks/mdsetup_ph/biobb_amber_CpHMD_notebook.
↪ipynb
```

ABC MD Setup tutorial

```
jupyter-notebook biobb_wf_amber_md_setup/notebooks/abcsetup/biobb_amber_ABC_setup.ipynb
```

1.1.3 Version

2021.2 Release

1.1.4 Copyright & Licensing

1.2 Protein MD Setup tutorial using BioExcel Building Blocks (biobb)

1.2.1 *–AmberTools package version–*

Based on the MDWeb Amber FULL MD Setup tutorial

This tutorial aims to illustrate the process of **setting up a simulation system** containing a **protein**, step by step, using the **BioExcel Building Blocks library (biobb)** wrapping the **AmberTools** utility from the **AMBER package**. The particular example used is the **Lysozyme** protein (PDB code 1AKI).

1.2.2 Settings

Biobb modules used

- biobb_io: Tools to fetch biomolecular data from public databases.
- biobb_amber: Tools to setup and run Molecular Dynamics simulations with AmberTools.
- biobb_analysis: Tools to analyse Molecular Dynamics trajectories.

Auxiliar libraries used

- nb_conda_kernels: Enables a Jupyter Notebook or JupyterLab application in one conda environment to access kernels for Python, R, and other languages found in other environments.
- nglview: Jupyter/IPython widget to interactively view molecular structures and trajectories in notebooks.
- ipywidgets: Interactive HTML widgets for Jupyter notebooks and the IPython kernel.
- plotly: Python interactive graphing library integrated in Jupyter notebooks.
- simpletraj: Lightweight coordinate-only trajectory reader based on code from GROMACS, MDAnalysis and VMD.

Conda Installation and Launch

```
git clone https://github.com/bioexcel/biobb_wf_amber_md_setup.git
cd biobb_wf_amber_md_setup
conda env create -f conda_env/environment.yml
conda activate biobb_AMBER_MDsetup_tutorials
jupyter-nbextension enable --py --user widgetsnbextension
jupyter-nbextension enable --py --user nglview
jupyter-notebook biobb_wf_amber_md_setup/notebooks/mdsetup/biobb_amber_setup_notebook.
↪ipynb
```

1.2.3 Pipeline steps

1. *Input Parameters*
2. *Fetching PDB Structure*
3. *Preparing PDB file for AMBER*
4. *Create Protein System Topology*
5. *Energetically Minimize the Structure*
6. *Create Solvent Box and Solvating the System*
7. *Adding Ions*
8. *Energetically Minimize the System*
9. *Heating the System*
10. *Equilibrate the System (NVT)*

11. *Equilibrate the System (NPT)*
12. *Free Molecular Dynamics Simulation*
13. *Post-processing and Visualizing Resulting 3D Trajectory*
14. *Output Files*
15. *Questions & Comments*

1.2.4 Input parameters

Input parameters needed:

- **pdbCode**: PDB code of the protein structure (e.g. 1AKI)

```
import nglview
import ipywidgets
import plotly
from plotly import subplots
import plotly.graph_objs as go

pdbCode = "1aki"
```

1.2.5 Fetching PDB structure

Downloading **PDB structure** with the **protein molecule** from the RCSB PDB database. Alternatively, a **PDB file** can be used as starting structure.

Building Blocks used:

- pdb from **biobb_io.api.pdb**

```
# Import module
from biobb_io.api.pdb import pdb

# Create properties dict and inputs/outputs
downloaded_pdb = pdbCode+'.pdb'

prop = {
    'pdb_code': pdbCode
}

#Create and launch bb
pdb(output_pdb_path=downloaded_pdb,
    properties=prop)
```

Visualizing 3D structure

Visualizing the downloaded/given **PDB structure** using **NGL**:

```
# Show protein
view = nglview.show_structure_file(downloaded_pdb)
view.add_representation(repr_type='ball+stick', selection='all')
view._remote_call('setSize', target='Widget', args=['','600px'])

view
```

1.2.6 Preparing PDB file for AMBER

Before starting a **protein MD setup**, it is always strongly recommended to take a look at the initial structure and try to identify important **properties** and also possible **issues**. These properties and issues can be serious, as for example the definition of **disulfide bridges**, the presence of a **non-standard aminoacids** or **ligands**, or **missing residues**. Other **properties** and **issues** might not be so serious, but they still need to be addressed before starting the **MD setup process**. **Missing hydrogen atoms**, presence of **alternate atomic location indicators** or **inserted residue codes** (see PDB file format specification) are examples of these not so crucial characteristics. Please visit the AMBER tutorial: Building Protein Systems in Explicit Solvent for more examples. **AmberTools** utilities from **AMBER MD package** contain a tool able to analyse **PDB files** and clean them for further usage, especially with the **AmberTools LEaP program**: the **pdb4amber tool**. The next step of the workflow is running this tool to analyse our **input PDB structure**.

For the particular **Lysosyme** example, the most important property that is identified by the **pdb4amber** utility is the presence of **disulfide bridges** in the structure. Those are marked changing the residue names **from CYS to CYX**, which is the code that **AMBER force fields** use to distinguish between cysteines forming or not forming **disulfide bridges**. This will be used in the following step to correctly form a **bond** between these cysteine residues.

We invite you to check what the tool does with different, more complex structures (e.g. PDB code 6N3V).

Building Blocks used:

- pdb4amber_run from **biobb_amber.pdb4amber.pdb4amber_run**

```
# Import module
from biobb_amber.pdb4amber.pdb4amber_run import pdb4amber_run

# Create prop dict and inputs/outputs
output_pdb4amber_path = 'structure.pdb4amber.pdb'

# Create and launch bb
pdb4amber_run(input_pdb_path=downloaded_pdb,
              output_pdb_path=output_pdb4amber_path)
```

1.2.7 Create protein system topology

Building AMBER topology corresponding to the protein structure.

IMPORTANT: the previous pdb4amber building block is changing the proper cysteines residue naming in the PDB file from CYS to CYX so that this step can automatically identify and add the disulfide bonds to the system topology.

The **force field** used in this tutorial is **ff14SB**, an evolution of the **ff99SB** force field with improved accuracy of protein side chains and backbone parameters. **Water** molecules type used in this tutorial is **tip3p**. Adding **side chain atoms** and **hydrogen atoms** if missing. Forming **disulfide bridges** according to the info added in the previous step.

Generating three output files:

- **AMBER structure** (PDB file)
- **AMBER topology** (AMBER Parmtop file)
- **AMBER coordinates** (AMBER Coordinate/Restart file)

Building Blocks used:

- leap_gen_top from **biobb_amber.leap.leap_gen_top**

```
# Import module
from biobb_amber.leap.leap_gen_top import leap_gen_top

# Create prop dict and inputs/outputs
output_pdb_path = 'structure.leap.pdb'
output_top_path = 'structure.leap.top'
output_crd_path = 'structure.leap.crd'

prop = {
    "forcefield" : ["protein.ff14SB"]
}

# Create and launch bb
leap_gen_top(input_pdb_path=output_pdb4amber_path,
             #input_pdb_path=downloaded_pdb,
             output_pdb_path=output_pdb_path,
             output_top_path=output_top_path,
             output_crd_path=output_crd_path,
             properties=prop)
```

Visualizing 3D structure

Visualizing the **PDB structure** using **NGL**. Try to identify the differences between the structure generated for the **system topology** and the **original one** (e.g. hydrogen atoms).

```
# Show protein
view = nglview.show_structure_file(output_pdb_path)
view.add_representation(repr_type='ball+stick', selection='all')
view._remote_call('setSize', target='Widget', args=['','600px'])

view
```

1.2.8 Energetically minimize the structure

Energetically minimize the **protein structure** (in vacuo) using the **sander tool** from the **AMBER MD package**. This step is **relaxing the structure**, usually **constrained**, especially when coming from an X-ray **crystal structure**.

The **miminization process** is done in two steps:

- *Step 1*: **Hydrogen** minimization, applying **position restraints** (50 Kcal/mol.$Å^{2}$) to the **protein heavy atoms**.
- *Step 2*: **System** minimization, applying **position restraints** (50 Kcal/mol.$Å^{2}$) to the **protein heavy atoms**.

Building Blocks used:

- sander_mdrun from **biobb_amber.sander.sander_mdrun**
- process_minout from **biobb_amber.process.process_minout**

Step 1: Minimize Hydrogens

Hydrogen minimization, applying **position restraints** (50 Kcal/mol.$Å^{2}$) to the **protein heavy atoms**.

```
# Import module
from biobb_amber.sander.sander_mdrun import sander_mdrun

# Create prop dict and inputs/outputs
output_h_min_traj_path = 'sander.h_min.x'
output_h_min_rst_path = 'sander.h_min.rst'
output_h_min_log_path = 'sander.h_min.log'

prop = {
    'simulation_type' : "min_vacuo",
    "mdin" : {
        'maxcyc' : 500,
        'ntpr' : 5,
        'ntr' : 1,
        'restraintmask' : '\":*&!@H=\"',
        'restraint_wt' : 50.0
    }
}

# Create and launch bb
sander_mdrun(input_top_path=output_top_path,
             input_crd_path=output_crd_path,
             input_ref_path=output_crd_path,
             output_traj_path=output_h_min_traj_path,
             output_rst_path=output_h_min_rst_path,
```

(continues on next page)

(continued from previous page)

```
            output_log_path=output_h_min_log_path,
            properties=prop)
```

Checking Energy Minimization results

Checking **energy minimization** results. Plotting **potential energy** along time during the **minimization process**.

```
# Import module
from biobb_amber.process.process_minout import process_minout

# Create prop dict and inputs/outputs
output_h_min_dat_path = 'sander.h_min.energy.dat'

prop = {
    "terms" : ['ENERGY']
}

# Create and launch bb
process_minout(input_log_path=output_h_min_log_path,
            output_dat_path=output_h_min_dat_path,
            properties=prop)
```

```
#Read data from file and filter energy values higher than 1000 Kj/mol^-1
with open(output_h_min_dat_path,'r') as energy_file:
    x,y = map(
        list,
        zip(*[
            (float(line.split()[0]),float(line.split()[1]))
            for line in energy_file
            if not line.startswith(("#","@"))
            if float(line.split()[1]) < 1000
        ])
    )

plotly.offline.init_notebook_mode(connected=True)

fig = {
    "data": [go.Scatter(x=x, y=y)],
    "layout": go.Layout(title="Energy Minimization",
                        xaxis=dict(title = "Energy Minimization Step"),
                        yaxis=dict(title = "Potential Energy kcal/mol")
                       )
}

plotly.offline.iplot(fig)
```

Step 2: Minimize the system

System minimization, applying **position restraints** (50 Kcal/mol.$Å^{2}$) to the **protein heavy atoms**.

```
# Import module
from biobb_amber.sander.sander_mdrun import sander_mdrun

# Create prop dict and inputs/outputs
output_n_min_traj_path = 'sander.n_min.x'
output_n_min_rst_path = 'sander.n_min.rst'
output_n_min_log_path = 'sander.n_min.log'

prop = {
    'simulation_type' : "min_vacuo",
    "mdin" : {
        'maxcyc' : 500,
        'ntpr' : 5,
        'ntr' : 1,
        'restraintmask' : '\":*&!@H=\"',
        'restraint_wt' : 50.0
    }
}

# Create and launch bb
sander_mdrun(input_top_path=output_top_path,
             input_crd_path=output_h_min_rst_path,
             input_ref_path=output_h_min_rst_path,
             output_traj_path=output_n_min_traj_path,
             output_rst_path=output_n_min_rst_path,
             output_log_path=output_n_min_log_path,
             properties=prop)
```

Checking Energy Minimization results

Checking **energy minimization** results. Plotting **potential energy** by time during the **minimization process**.

```
# Import module
from biobb_amber.process.process_minout import process_minout

# Create prop dict and inputs/outputs
output_n_min_dat_path = 'sander.n_min.energy.dat'

prop = {
    "terms" : ['ENERGY']
}

# Create and launch bb
process_minout(input_log_path=output_n_min_log_path,
             output_dat_path=output_n_min_dat_path,
             properties=prop)
```

```
#Read data from file and filter energy values higher than 1000 Kj/mol^-1
with open(output_n_min_dat_path,'r') as energy_file:
    x,y = map(
        list,
        zip(*[
            (float(line.split()[0]),float(line.split()[1]))
            for line in energy_file
            if not line.startswith(("#","@"))
            if float(line.split()[1]) < 1000
        ])
    )

plotly.offline.init_notebook_mode(connected=True)

fig = {
    "data": [go.Scatter(x=x, y=y)],
    "layout": go.Layout(title="Energy Minimization",
                        xaxis=dict(title = "Energy Minimization Step"),
                        yaxis=dict(title = "Potential Energy kcal/mol")
                       )
}

plotly.offline.iplot(fig)
```

1.2.9 Create solvent box and solvating the system

Define the unit cell for the **protein structure MD system** to fill it with water molecules. A **truncated octahedron box** is used to define the unit cell, with a **distance from the protein to the box edge of 9.0 Angstroms**. The solvent type used is the default **TIP3P** water model, a generic 3-point solvent model.

Building Blocks used:

- amber_to_pdb from **biobb_amber.ambpdb.amber_to_pdb**
- leap_solvate from **biobb_amber.leap.leap_solvate**

Getting minimized structure

Getting the result of the **energetic minimization** and converting it to **PDB format** to be then used as input for the **water box generation**. This is achieved by converting from **AMBER topology + coordinates** files to a **PDB file** using the **ambpdb** tool from the **AMBER MD package**.

```
# Import module
from biobb_amber.ambpdb.amber_to_pdb import amber_to_pdb

# Create prop dict and inputs/outputs
output_ambpdb_path = 'structure.ambpdb.pdb'

# Create and launch bb
amber_to_pdb(input_top_path=output_top_path,
             input_crd_path=output_h_min_rst_path,
             output_pdb_path=output_ambpdb_path
             )
```

Create water box

Define the **unit cell** for the **protein structure MD system** and fill it with **water molecules**.

```
# Import module
from biobb_amber.leap.leap_solvate import leap_solvate

# Create prop dict and inputs/outputs
output_solv_pdb_path = 'structure.solv.pdb'
output_solv_top_path = 'structure.solv.parmtop'
output_solv_crd_path = 'structure.solv.crd'

prop = {
    "forcefield" : ["protein.ff14SB"],
    "water_type": "TIP3PBOX",
    "distance_to_molecule": "9.0",
    "box_type": "truncated_octahedron"
}

# Create and launch bb
leap_solvate(input_pdb_path=output_ambpdb_path,
             output_pdb_path=output_solv_pdb_path,
             output_top_path=output_solv_top_path,
             output_crd_path=output_solv_crd_path,
             properties=prop)
```

1.2.10 Adding ions

Neutralizing the system and adding an additional **ionic concentration** using the **leap tool** from the **AMBER MD package**. Using **Sodium (Na+)** and **Chloride (Cl-)** counterions and an **additional ionic concentration** of 150mM.

Building Blocks used:

- leap_add_ions from **biobb_amber.leap.leap_add_ions**

```
# Import module
from biobb_amber.leap.leap_add_ions import leap_add_ions

# Create prop dict and inputs/outputs
output_ions_pdb_path = 'structure.ions.pdb'
output_ions_top_path = 'structure.ions.parmtop'
output_ions_crd_path = 'structure.ions.crd'

prop = {
    "forcefield" : ["protein.ff14SB"],
    "neutralise" : True,
    "positive_ions_type": "Na+",
    "negative_ions_type": "Cl-",
    "ionic_concentration" : 150, # 150mM
    "box_type": "truncated_octahedron"
}

# Create and launch bb
leap_add_ions(input_pdb_path=output_solv_pdb_path,
            output_pdb_path=output_ions_pdb_path,
            output_top_path=output_ions_top_path,
            output_crd_path=output_ions_crd_path,
            properties=prop)
```

Visualizing 3D structure

Visualizing the **protein system** with the newly added **solvent box** and **counterions** using **NGL**. Note the **truncated octahedron box** filled with **water molecules** surrounding the **protein structure**, as well as the randomly placed **positive** (Na+, blue) and **negative** (Cl-, gray) **counterions**.

```
# Show protein
view = nglview.show_structure_file(output_ions_pdb_path)
view.clear_representations()
view.add_representation(repr_type='cartoon', selection='protein')
view.add_representation(repr_type='ball+stick', selection='solvent')
view.add_representation(repr_type='spacefill', selection='Cl- Na+')
view._remote_call('setSize', target='Widget', args=['','600px'])

view
```

1.2.11 Energetically minimize the system

Energetically minimize the **system** (protein structure + solvent + ions) using the **sander tool** from the **AMBER MD package**. **Restraining heavy atoms** with a force constant of 15 Kcal/mol.$Å^{2}$ to their initial positions.

- *Step 1*: Energetically minimize the **system** through 500 minimization cycles.
- *Step 2*: Checking **energy minimization** results. Plotting energy by time during the **minimization** process.

Building Blocks used:

- sander_mdrun from **biobb_amber.sander.sander_mdrun**
- process_minout from **biobb_amber.process.process_minout**

Step 1: Running Energy Minimization

The **minimization** type of the **simulation_type property** contains the main default parameters to run an **energy minimization**:

- imin = 1 ; Minimization flag, perform an energy minimization.
- maxcyc = 500; The maximum number of cycles of minimization.
- ntb = 1; Periodic boundaries: constant volume.
- ntmin = 2; Minimization method: steepest descent.

In this particular example, the method used to run the **energy minimization** is the default **steepest descent**, with a **maximum number of 500 cycles** and **periodic conditions**.

```
# Import module
from biobb_amber.sander.sander_mdrun import sander_mdrun

# Create prop dict and inputs/outputs
output_min_traj_path = 'sander.min.x'
output_min_rst_path = 'sander.min.rst'
output_min_log_path = 'sander.min.log'

prop = {
    "simulation_type" : "minimization",
    "mdin" : {
        'maxcyc' : 300, # Reducing the number of minimization steps for the sake of time
        'ntr' : 1,      # Overwritting restrain parameter
        'restraintmask' : '\"!:WAT,Cl-,Na+\"',      # Restraining solute
        'restraint_wt' : 50.0                       # With a force constant of 50 Kcal/
↪mol*A2
    }
}

# Create and launch bb
sander_mdrun(input_top_path=output_ions_top_path,
             input_crd_path=output_ions_crd_path,
```

(continues on next page)

(continued from previous page)

```
            input_ref_path=output_ions_crd_path,
            output_traj_path=output_min_traj_path,
            output_rst_path=output_min_rst_path,
            output_log_path=output_min_log_path,
            properties=prop)
```

Step 2: Checking Energy Minimization results

Checking **energy minimization** results. Plotting **potential energy** along time during the **minimization process**.

```
# Import module
from biobb_amber.process.process_minout import process_minout

# Create prop dict and inputs/outputs
output_dat_path = 'sander.min.energy.dat'

prop = {
    "terms" : ['ENERGY']
}

# Create and launch bb
process_minout(input_log_path=output_min_log_path,
            output_dat_path=output_dat_path,
            properties=prop)
```

```
#Read data from file and filter energy values higher than 1000 Kj/mol^-1
with open(output_dat_path,'r') as energy_file:
    x,y = map(
        list,
        zip(*[
            (float(line.split()[0]),float(line.split()[1]))
            for line in energy_file
            if not line.startswith(("#","@"))
            if float(line.split()[1]) < 1000
        ])
    )

plotly.offline.init_notebook_mode(connected=True)

fig = {
    "data": [go.Scatter(x=x, y=y)],
    "layout": go.Layout(title="Energy Minimization",
                        xaxis=dict(title = "Energy Minimization Step"),
                        yaxis=dict(title = "Potential Energy kcal/mol")
                       )
}

plotly.offline.iplot(fig)
```

1.2.12 Heating the system

Warming up the **prepared system** using the **sander tool** from the **AMBER MD package**. Going from 0 to the desired **temperature**, in this particular example, 300K. **Solute atoms restrained** (force constant of 10 Kcal/mol). Length 5ps.

- *Step 1*: Warming up the **system** through 500 MD steps.
- *Step 2*: Checking results for the **system warming up**. Plotting **temperature** along time during the **heating** process.

Building Blocks used:

- sander_mdrun from **biobb_amber.sander.sander_mdrun**
- process_mdout from **biobb_amber.process.process_mdout**

Step 1: Warming up the system

The **heat** type of the **simulation_type property** contains the main default parameters to run a **system warming up**:

- imin = 0; Run MD (no minimization)
- ntx = 5; Read initial coords and vels from restart file
- cut = 10.0; Cutoff for non bonded interactions in Angstroms
- ntr = 0; No restrained atoms
- ntc = 2; SHAKE for constraining length of bonds involving Hydrogen atoms
- ntf = 2; Bond interactions involving H omitted
- ntt = 3; Constant temperature using Langevin dynamics
- ig = -1; Seed for pseudo-random number generator
- ioutfm = 1; Write trajectory in netcdf format
- iwrap = 1; Wrap coords into primary box
- nstlim = 5000; Number of MD steps
- dt = 0.002; Time step (in ps)
- tempi = 0.0; Initial temperature (0 K)
- temp0 = 300.0; Final temperature (300 K)
- irest = 0; No restart from previous simulation
- ntb = 1; Periodic boundary conditions at constant volume
- gamma_ln = 1.0; Collision frequency for Langevin dynamics (in 1/ps)

In this particular example, the **heating** of the system is done in **2500 steps** (5ps) and is going **from 0K to 300K** (note that the number of steps has been reduced in this tutorial, for the sake of time).

```
# Import module
from biobb_amber.sander.sander_mdrun import sander_mdrun

# Create prop dict and inputs/outputs
output_heat_traj_path = 'sander.heat.netcdf'
output_heat_rst_path = 'sander.heat.rst'
output_heat_log_path = 'sander.heat.log'

prop = {
    "simulation_type" : "heat",
    "mdin" : {
        'nstlim' : 2500, # Reducing the number of steps for the sake of time (5ps)
        'ntr' : 1,       # Overwritting restrain parameter
        'restraintmask' : '\"!:WAT,Cl-,Na+\"',      # Restraining solute
        'restraint_wt' : 10.0                       # With a force constant of 10 Kcal/
↪mol*A2
    }
}

# Create and launch bb
sander_mdrun(input_top_path=output_ions_top_path,
             input_crd_path=output_min_rst_path,
             input_ref_path=output_min_rst_path,
             output_traj_path=output_heat_traj_path,
             output_rst_path=output_heat_rst_path,
             output_log_path=output_heat_log_path,
             properties=prop)
```

Step 2: Checking results from the system warming up

Checking **system warming up** output. Plotting **temperature** along time during the **heating process**.

```
# Import module
from biobb_amber.process.process_mdout import process_mdout

# Create prop dict and inputs/outputs
output_dat_heat_path = 'sander.md.temp.dat'

prop = {
    "terms" : ['TEMP']
}

# Create and launch bb
process_mdout(input_log_path=output_heat_log_path,
             output_dat_path=output_dat_heat_path,
             properties=prop)
```

```
#Read data from file and filter energy values higher than 1000 Kj/mol^-1
with open(output_dat_heat_path,'r') as energy_file:
    x,y = map(
```

(continues on next page)

(continued from previous page)

```
        list,
        zip(*[
            (float(line.split()[0]),float(line.split()[1]))
            for line in energy_file
            if not line.startswith(("#","@"))
            if float(line.split()[1]) < 1000
        ])
    )

plotly.offline.init_notebook_mode(connected=True)

fig = {
    "data": [go.Scatter(x=x, y=y)],
    "layout": go.Layout(title="Heating process",
                        xaxis=dict(title = "Heating Step (ps)"),
                        yaxis=dict(title = "Temperature (K)")
                       )
}

plotly.offline.iplot(fig)
```

1.2.13 Equilibrate the system (NVT)

Equilibrate the **protein system** in **NVT ensemble** (constant Number of particles, Volume and Temperature). Protein **heavy atoms** will be restrained using position restraining forces: movement is permitted, but only after overcoming a substantial energy penalty. The utility of position restraints is that they allow us to equilibrate our solvent around our protein, without the added variable of structural changes in the protein.

- *Step 1*: Equilibrate the **protein system** with **NVT** ensemble.
- *Step 2*: Checking **NVT Equilibration** results. Plotting **system temperature** by time during the **NVT equilibration** process.

Building Blocks used:

- sander_mdrun from **biobb_amber.sander.sander_mdrun**
- process_mdout from **biobb_amber.process.process_mdout**

Step 1: Equilibrating the system (NVT)

The **nvt** type of the **simulation_type property** contains the main default parameters to run a **system equilibration in NVT ensemble**:

- imin = 0; Run MD (no minimization)
- ntx = 5; Read initial coords and vels from restart file
- cut = 10.0; Cutoff for non bonded interactions in Angstroms
- ntr = 0; No restrained atoms
- ntc = 2; SHAKE for constraining length of bonds involving Hydrogen atoms
- ntf = 2; Bond interactions involving H omitted
- ntt = 3; Constant temperature using Langevin dynamics
- ig = -1; Seed for pseudo-random number generator
- ioutfm = 1; Write trajectory in netcdf format
- iwrap = 1; Wrap coords into primary box
- nstlim = 5000; Number of MD steps
- dt = 0.002; Time step (in ps)
- irest = 1; Restart previous simulation
- ntb = 1; Periodic boundary conditions at constant volume
- gamma_ln = 5.0; Collision frequency for Langevin dynamics (in 1/ps)

In this particular example, the **NVT equilibration** of the system is done in **500 steps** (note that the number of steps has been reduced in this tutorial, for the sake of time).

```
# Import module
from biobb_amber.sander.sander_mdrun import sander_mdrun

# Create prop dict and inputs/outputs
output_nvt_traj_path = 'sander.nvt.netcdf'
output_nvt_rst_path = 'sander.nvt.rst'
output_nvt_log_path = 'sander.nvt.log'

prop = {
    "simulation_type" : 'nvt',
    "mdin" : {
        'nstlim' : 500, # Reducing the number of steps for the sake of time (1ps)
        'ntr' : 1,      # Overwritting restrain parameter
        'restraintmask' : '\"!:WAT,Cl-,Na+ & !@H=\"',      # Restraining solute heavy␣
↪atoms
        'restraint_wt' : 5.0                               # With a force constant of 5␣
↪Kcal/mol*A2
    }
}

# Create and launch bb
sander_mdrun(input_top_path=output_ions_top_path,
             input_crd_path=output_heat_rst_path,
```

(continues on next page)

(continued from previous page)

```
            input_ref_path=output_heat_rst_path,
            output_traj_path=output_nvt_traj_path,
            output_rst_path=output_nvt_rst_path,
            output_log_path=output_nvt_log_path,
            properties=prop)
```

Step 2: Checking NVT Equilibration results

Checking **NVT Equilibration** results. Plotting **system temperature** by time during the NVT equilibration process.

```
# Import module
from biobb_amber.process.process_mdout import process_mdout

# Create prop dict and inputs/outputs
output_dat_nvt_path = 'sander.md.nvt.temp.dat'

prop = {
    "terms" : ['TEMP']
}

# Create and launch bb
process_mdout(input_log_path=output_nvt_log_path,
            output_dat_path=output_dat_nvt_path,
            properties=prop)
```

```
#Read data from file and filter energy values higher than 1000 Kj/mol^-1
with open(output_dat_nvt_path,'r') as energy_file:
    x,y = map(
        list,
        zip(*[
            (float(line.split()[0]),float(line.split()[1]))
            for line in energy_file
            if not line.startswith(("#","@"))
            if float(line.split()[1]) < 1000
        ])
    )

plotly.offline.init_notebook_mode(connected=True)

fig = {
    "data": [go.Scatter(x=x, y=y)],
    "layout": go.Layout(title="NVT equilibration",
                        xaxis=dict(title = "Equilibration Step (ps)"),
                        yaxis=dict(title = "Temperature (K)")
                       )
}

plotly.offline.iplot(fig)
```

1.2.14 Equilibrate the system (NPT)

Equilibrate the **protein system** in **NPT ensemble** (constant Number of particles, Pressure and Temperature). Protein **heavy atoms** will be restrained using position restraining forces: movement is permitted, but only after overcoming a substantial energy penalty. The utility of position restraints is that they allow us to equilibrate our solvent around our protein, without the added variable of structural changes in the protein.

- *Step 1*: Equilibrate the **protein system** with **NPT** ensemble.
- *Step 2*: Checking **NPT Equilibration** results. Plotting **system pressure and density** by time during the **NVT equilibration** process.

Building Blocks used:

- sander_mdrun from **biobb_amber.sander.sander_mdrun**
- process_mdout from **biobb_amber.process.process_mdout**

Step 1: Equilibrating the system (NPT)

The **npt** type of the **simulation_type property** contains the main default parameters to run a **system equilibration in NPT ensemble**:

- imin = 0; Run MD (no minimization)
- ntx = 5; Read initial coords and vels from restart file
- cut = 10.0; Cutoff for non bonded interactions in Angstroms
- ntr = 0; No restrained atoms
- ntc = 2; SHAKE for constraining length of bonds involving Hydrogen atoms
- ntf = 2; Bond interactions involving H omitted
- ntt = 3; Constant temperature using Langevin dynamics
- ig = -1; Seed for pseudo-random number generator
- ioutfm = 1; Write trajectory in netcdf format
- iwrap = 1; Wrap coords into primary box
- nstlim = 5000; Number of MD steps
- dt = 0.002; Time step (in ps)
- irest = 1; Restart previous simulation
- gamma_ln = 5.0; Collision frequency for Langevin dynamics (in 1/ps)
- pres0 = 1.0; Reference pressure
- ntp = 1; Constant pressure dynamics: md with isotropic position scaling

- taup = 2.0; Pressure relaxation time (in ps)

In this particular example, the **NPT equilibration** of the system is done in **500 steps** (note that the number of steps has been reduced in this tutorial, for the sake of time).

```
# Import module
from biobb_amber.sander.sander_mdrun import sander_mdrun

# Create prop dict and inputs/outputs
output_npt_traj_path = 'sander.npt.netcdf'
output_npt_rst_path = 'sander.npt.rst'
output_npt_log_path = 'sander.npt.log'

prop = {
    "simulation_type" : 'npt',
    "mdin" : {
        'nstlim' : 500, # Reducing the number of steps for the sake of time (1ps)
        'ntr' : 1,      # Overwritting restrain parameter
        'restraintmask' : '\"!:WAT,Cl-,Na+ & !@H=\"',      # Restraining solute heavy␣
↪atoms
        'restraint_wt' : 2.5                               # With a force constant of 2.
↪5 Kcal/mol*A2
    }
}

# Create and launch bb
sander_mdrun(input_top_path=output_ions_top_path,
             input_crd_path=output_nvt_rst_path,
             input_ref_path=output_nvt_rst_path,
             output_traj_path=output_npt_traj_path,
             output_rst_path=output_npt_rst_path,
             output_log_path=output_npt_log_path,
             properties=prop)
```

Step 2: Checking NPT Equilibration results

Checking **NPT Equilibration** results. Plotting **system pressure and density** by time during the **NPT equilibration** process.

```
# Import module
from biobb_amber.process.process_mdout import process_mdout

# Create prop dict and inputs/outputs
output_dat_npt_path = 'sander.md.npt.dat'

prop = {
    "terms" : ['PRES','DENSITY']
}

# Create and launch bb
process_mdout(input_log_path=output_npt_log_path,
```

(continues on next page)

(continued from previous page)

```
            output_dat_path=output_dat_npt_path,
            properties=prop)
```

```
# Read pressure and density data from file
with open(output_dat_npt_path,'r') as pd_file:
    x,y,z = map(
        list,
        zip(*[
            (float(line.split()[0]),float(line.split()[1]),float(line.split()[2]))
            for line in pd_file
            if not line.startswith(("#","@"))
        ])
    )

plotly.offline.init_notebook_mode(connected=True)

trace1 = go.Scatter(
    x=x,y=y
)
trace2 = go.Scatter(
    x=x,y=z
)

fig = subplots.make_subplots(rows=1, cols=2, print_grid=False)

fig.append_trace(trace1, 1, 1)
fig.append_trace(trace2, 1, 2)

fig['layout']['xaxis1'].update(title='Time (ps)')
fig['layout']['xaxis2'].update(title='Time (ps)')
fig['layout']['yaxis1'].update(title='Pressure (bar)')
fig['layout']['yaxis2'].update(title='Density (Kg*m^-3)')

fig['layout'].update(title='Pressure and Density during NPT Equilibration')
fig['layout'].update(showlegend=False)

plotly.offline.iplot(fig)
```

1.2.15 Free Molecular Dynamics Simulation

Upon completion of the **two equilibration phases (NVT and NPT)**, the system is now well-equilibrated at the desired temperature and pressure. The **position restraints** can now be released. The last step of the **protein** MD setup is a short, **free MD simulation**, to ensure the robustness of the system.

- *Step 1*: Run short MD simulation of the **protein system**.
- *Step 2*: Checking results for the final step of the setup process, the **free MD run**. Plotting **Root Mean Square deviation (RMSd)** and **Radius of Gyration (Rgyr)** by time during the **free MD run** step.

Building Blocks used:

- sander_mdrun from **biobb_amber.sander.sander_mdrun**
- process_mdout from **biobb_amber.process.process_mdout**
- cpptraj_rms from **biobb_analysis.cpptraj.cpptraj_rms**
- cpptraj_rgyr from **biobb_analysis.cpptraj.cpptraj_rgyr**

Step 1: Creating portable binary run file to run a free MD simulation

The **free** type of the **simulation_type property** contains the main default parameters to run an **unrestrained MD simulation**:

- imin = 0; Run MD (no minimization)
- ntx = 5; Read initial coords and vels from restart file
- cut = 10.0; Cutoff for non bonded interactions in Angstroms
- ntr = 0; No restrained atoms
- ntc = 2; SHAKE for constraining length of bonds involving Hydrogen atoms
- ntf = 2; Bond interactions involving H omitted
- ntt = 3; Constant temperature using Langevin dynamics
- ig = -1; Seed for pseudo-random number generator
- ioutfm = 1; Write trajectory in netcdf format
- iwrap = 1; Wrap coords into primary box
- nstlim = 5000; Number of MD steps
- dt = 0.002; Time step (in ps)

In this particular example, a short, **5ps-length** simulation (2500 steps) is run, for the sake of time.

```
# Import module
from biobb_amber.sander.sander_mdrun import sander_mdrun

# Create prop dict and inputs/outputs
output_free_traj_path = 'sander.free.netcdf'
output_free_rst_path = 'sander.free.rst'
```

(continues on next page)

(continued from previous page)

```
output_free_log_path = 'sander.free.log'

prop = {
    "simulation_type" : 'free',
    "mdin" : {
        'nstlim' : 2500, # Reducing the number of steps for the sake of time (5ps)
        'ntwx' : 500  # Print coords to trajectory every 500 steps (1 ps)
    }
}

# Create and launch bb
sander_mdrun(input_top_path=output_ions_top_path,
             input_crd_path=output_npt_rst_path,
             output_traj_path=output_free_traj_path,
             output_rst_path=output_free_rst_path,
             output_log_path=output_free_log_path,
             properties=prop)
```

Step 2: Checking free MD simulation results

Checking results for the final step of the setup process, the **free MD run**. Plotting **Root Mean Square deviation (RMSd)** and **Radius of Gyration (Rgyr)** by time during the **free MD run** step. **RMSd** against the **experimental structure** (input structure of the pipeline) and against the **minimized and equilibrated structure** (output structure of the NPT equilibration step).

```
# cpptraj_rms: Computing Root Mean Square deviation to analyse structural stability
#              RMSd against minimized and equilibrated snapshot (backbone atoms)

# Import module
from biobb_analysis.ambertools.cpptraj_rms import cpptraj_rms

# Create prop dict and inputs/outputs
output_rms_first = pdbCode+'_rms_first.dat'

prop = {
    'mask': 'backbone',
    'reference': 'first'
}

# Create and launch bb
cpptraj_rms(input_top_path=output_ions_top_path,
             input_traj_path=output_free_traj_path,
             output_cpptraj_path=output_rms_first,
             properties=prop)
```

```
# cpptraj_rms: Computing Root Mean Square deviation to analyse structural stability
#              RMSd against experimental structure (backbone atoms)

# Import module
```

(continues on next page)

(continued from previous page)

```
from biobb_analysis.ambertools.cpptraj_rms import cpptraj_rms

# Create prop dict and inputs/outputs
output_rms_exp = pdbCode+'_rms_exp.dat'

prop = {
    'mask': 'backbone',
    'reference': 'experimental'
}

# Create and launch bb
cpptraj_rms(input_top_path=output_ions_top_path,
            input_traj_path=output_free_traj_path,
            output_cpptraj_path=output_rms_exp,
            input_exp_path=downloaded_pdb,
            properties=prop)
```

```
# Read RMS vs first snapshot data from file
with open(output_rms_first,'r') as rms_first_file:
    x,y = map(
        list,
        zip(*[
            (float(line.split()[0]),float(line.split()[1]))
            for line in rms_first_file
            if not line.startswith(("#","@"))
        ])
    )

# Read RMS vs experimental structure data from file
with open(output_rms_exp,'r') as rms_exp_file:
    x2,y2 = map(
        list,
        zip(*[
            (float(line.split()[0]),float(line.split()[1]))
            for line in rms_exp_file
            if not line.startswith(("#","@"))
        ])
    )

trace1 = go.Scatter(
    x = x,
    y = y,
    name = 'RMSd vs first'
)

trace2 = go.Scatter(
    x = x,
    y = y2,
    name = 'RMSd vs exp'
)

data = [trace1, trace2]
```

(continues on next page)

(continued from previous page)

```
plotly.offline.init_notebook_mode(connected=True)

fig = {
    "data": data,
    "layout": go.Layout(title="RMSd during free MD Simulation",
                        xaxis=dict(title = "Time (ps)"),
                        yaxis=dict(title = "RMSd (Angstrom)")
                       )
}

plotly.offline.iplot(fig)
```

```
# cpptraj_rgyr: Computing Radius of Gyration to measure the protein compactness during␣
↪the free MD simulation

# Import module
from biobb_analysis.ambertools.cpptraj_rgyr import cpptraj_rgyr

# Create prop dict and inputs/outputs
output_rgyr = pdbCode+'_rgyr.dat'

prop = {
    'mask': 'backbone'
}

# Create and launch bb
cpptraj_rgyr(input_top_path=output_ions_top_path,
             input_traj_path=output_free_traj_path,
             output_cpptraj_path=output_rgyr,
             properties=prop)
```

```
# Read Rgyr data from file
with open(output_rgyr,'r') as rgyr_file:
    x,y = map(
        list,
        zip(*[
            (float(line.split()[0]),float(line.split()[1]))
            for line in rgyr_file
            if not line.startswith(("#","@"))
        ])
    )

plotly.offline.init_notebook_mode(connected=True)

fig = {
    "data": [go.Scatter(x=x, y=y)],
    "layout": go.Layout(title="Radius of Gyration",
                        xaxis=dict(title = "Time (ps)"),
                        yaxis=dict(title = "Rgyr (Angstrom)")
```

(continues on next page)

(continued from previous page)

```
                        )
}

plotly.offline.iplot(fig)
```

1.2.16 Post-processing and Visualizing resulting 3D trajectory

Post-processing and Visualizing the **protein system** MD setup **resulting trajectory** using **NGL**

- *Step 1*: *Imaging* the resulting trajectory, **stripping out water molecules and ions** and **correcting periodicity issues**.
- *Step 2*: Visualizing the *imaged* trajectory using the *dry* structure as a **topology**.

Building Blocks used:

- cpptraj_image from **biobb_analysis.cpptraj.cpptraj_image**

Step 1: *Imaging* the resulting trajectory.

Stripping out **water molecules and ions** and **correcting periodicity issues**

```
# cpptraj_image: "Imaging" the resulting trajectory
#                Removing water molecules and ions from the resulting structure

# Import module
from biobb_analysis.ambertools.cpptraj_image import cpptraj_image

# Create prop dict and inputs/outputs
output_imaged_traj = pdbCode+'_imaged_traj.trr'

prop = {
    'mask': 'solute',
    'format': 'trr'
}

# Create and launch bb
cpptraj_image(input_top_path=output_ions_top_path,
            input_traj_path=output_free_traj_path,
            output_cpptraj_path=output_imaged_traj,
            properties=prop)
```

Step 2: Visualizing the generated dehydrated trajectory.

Using the **imaged trajectory** (output of the *Post-processing step 1*) with the **dry structure** (output of the *Structure energy minimization - step 5*) as a topology.

```
# Show trajectory
view = nglview.show_simpletraj(nglview.SimpletrajTrajectory(output_imaged_traj, output_
↪ambpdb_path), gui=True)
view
```

1.2.17 Output files

Important **Output files** generated:

- structure.ions.pdb: **System structure** of the MD setup protocol. Structure generated during the MD setup and used in the MD simulation. With hydrogen atoms, solvent box and counterions.
- sander.free.netcdf: **Final trajectory** of the MD setup protocol.
- sander.free.rst: **Final checkpoint file**, with information about the state of the simulation. It can be used to **restart** or **continue** a MD simulation.
- structure.ions.parmtop: **Final topology** of the MD system in AMBER Parm7 format.

Analysis (MD setup check) output files generated:

- 1aki_rms_first.dat: **Root Mean Square deviation (RMSd)** against **minimized and equilibrated structure** of the final **free MD run step**.
- 1aki_rms_exp.dat: **Root Mean Square deviation (RMSd)** against **experimental structure** of the final **free MD run step**.
- 1aki_rgyr.dat: **Radius of Gyration** of the final **free MD run step** of the **setup pipeline**.

1.2.18 Questions & Comments

Questions, issues, suggestions and comments are really welcome!

- GitHub issues:
  - https://github.com/bioexcel/biobb
- BioExcel forum:
  - https://ask.bioexcel.eu/c/BioExcel-Building-Blocks-library

1.3 Protein-ligand complex MD Setup tutorial using BioExcel Building Blocks (biobb)

1.3.1 *–AmberTools package version–*

Based on the MDWeb Amber FULL MD Setup tutorial

This tutorial aims to illustrate the process of **setting up a simulation system** containing a **protein in complex with a ligand**, step by step, using the **BioExcel Building Blocks library (biobb)** wrapping the **AmberTools** utility from the **AMBER package**. The particular example used is the **T4 lysozyme** protein (PDB code 3HTB) with two residue modifications ***L99A/M102Q*** complexed with the small ligand **2-propylphenol** (3-letter code JZ4).

1.3.2 Settings

Biobb modules used

- biobb_io: Tools to fetch biomolecular data from public databases.
- biobb_amber: Tools to setup and run Molecular Dynamics simulations with AmberTools.
- biobb_structure_utils: Tools to modify or extract information from a PDB structure file.
- biobb_analysis: Tools to analyse Molecular Dynamics trajectories.
- biobb_chemistry: Tools to to perform chemical conversions.

Auxiliar libraries used

- nb_conda_kernels: Enables a Jupyter Notebook or JupyterLab application in one conda environment to access kernels for Python, R, and other languages found in other environments.
- nglview: Jupyter/IPython widget to interactively view molecular structures and trajectories in notebooks.
- ipywidgets: Interactive HTML widgets for Jupyter notebooks and the IPython kernel.
- plotly: Python interactive graphing library integrated in Jupyter notebooks.
- simpletraj: Lightweight coordinate-only trajectory reader based on code from GROMACS, MDAnalysis and VMD.

Conda Installation and Launch

```
git clone https://github.com/bioexcel/biobb_wf_amber_md_setup.git
cd biobb_wf_amber_md_setup
conda env create -f conda_env/environment.yml
conda activate biobb_AMBER_MDsetup_tutorials
jupyter-nbextension enable --py --user widgetsnbextension
jupyter-nbextension enable --py --user nglview
jupyter-notebook biobb_wf_amber_md_setup/notebooks/mdsetup_lig/biobb_amber_complex_setup_
↪notebook.ipynb
```

1.3.3 Pipeline steps

1. *Input Parameters*
2. *Fetching PDB Structure*
3. *Preparing PDB file for AMBER*
4. *Create ligand system topology*
5. *Create Protein-Ligand Complex System Topology*
6. *Energetically Minimize the Structure*
7. *Create Solvent Box and Solvating the System*
8. *Adding Ions*
9. *Energetically Minimize the System*
10. *Heating the System*
11. *Equilibrate the System (NVT)*
12. *Equilibrate the System (NPT)*
13. *Free Molecular Dynamics Simulation*
14. *Post-processing and Visualizing Resulting 3D Trajectory*
15. *Output Files*
16. *Questions & Comments*

1.3.4 Input parameters

Input parameters needed:

- **pdbCode**: PDB code of the protein structure (e.g. 3HTB)
- **ligandCode**: 3-letter code of the ligand (e.g. JZ4)
- **mol_charge**: Charge of the ligand (e.g. 0)

```
import nglview
import ipywidgets
import plotly
from plotly import subplots
import plotly.graph_objs as go

pdbCode = "3htb"
ligandCode = "JZ4"
mol_charge = 0
```

1.3.5 Fetching PDB structure

Downloading **PDB structure** with the **protein molecule** from the RCSB PDB database. Alternatively, a **PDB file** can be used as starting structure. Stripping from the **downloaded structure** any **crystallographic water** molecule or **heteroatom**.

Building Blocks used:

- pdb from **biobb_io.api.pdb**
- remove_pdb_water from **biobb_structure_utils.utils.remove_pdb_water**
- remove_ligand from **biobb_structure_utils.utils.remove_ligand**

```
# Import module
from biobb_io.api.pdb import pdb

# Create properties dict and inputs/outputs
downloaded_pdb = pdbCode+'.pdb'

prop = {
    'pdb_code': pdbCode,
    'filter': False
}

#Create and launch bb
pdb(output_pdb_path=downloaded_pdb,
    properties=prop)
```

```
# Show protein
view = nglview.show_structure_file(downloaded_pdb)
view.clear_representations()
view.add_representation(repr_type='cartoon', selection='protein', color='sstruc')
view.add_representation(repr_type='ball+stick', radius='0.1', selection='water')
view.add_representation(repr_type='ball+stick', radius='0.5', selection='ligand')
view.add_representation(repr_type='ball+stick', radius='0.5', selection='ion')
view._remote_call('setSize', target='Widget', args=['','600px'])
view
```

```
# Import module
from biobb_structure_utils.utils.remove_pdb_water import remove_pdb_water

# Create properties dict and inputs/outputs
nowat_pdb = pdbCode+'.nowat.pdb'

#Create and launch bb
remove_pdb_water(input_pdb_path=downloaded_pdb,
    output_pdb_path=nowat_pdb)
```

```
# Import module
from biobb_structure_utils.utils.remove_ligand import remove_ligand

# Removing PO4 ligands:

# Create properties dict and inputs/outputs
nopo4_pdb = pdbCode+'.noPO4.pdb'

prop = {
    'ligand' : 'PO4'
}

#Create and launch bb
remove_ligand(input_structure_path=nowat_pdb,
    output_structure_path=nopo4_pdb,
    properties=prop)

# Removing BME ligand:

# Create properties dict and inputs/outputs
nobme_pdb = pdbCode+'.noBME.pdb'

prop = {
    'ligand' : 'BME'
}

#Create and launch bb
remove_ligand(input_structure_path=nopo4_pdb,
    output_structure_path=nobme_pdb,
    properties=prop)
```

1.3.6 Visualizing 3D structure

```
# Show protein
view = nglview.show_structure_file(nobme_pdb)
view.clear_representations()
view.add_representation(repr_type='cartoon', selection='protein', color='sstruc')
view.add_representation(repr_type='ball+stick', radius='0.5', selection='hetero')
view._remote_call('setSize', target='Widget', args=['','600px'])
view
```

1.3.7 Preparing PDB file for AMBER

Before starting a **protein MD setup**, it is always strongly recommended to take a look at the initial structure and try to identify important **properties** and also possible **issues**. These properties and issues can be serious, as for example the definition of **disulfide bridges**, the presence of a **non-standard aminoacids** or **ligands**, or **missing residues**. Other **properties** and **issues** might not be so serious, but they still need to be addressed before starting the **MD setup process**. **Missing hydrogen atoms**, presence of **alternate atomic location indicators** or **inserted residue codes** (see PDB file format specification) are examples of these not so crucial characteristics. Please visit the AMBER tutorial: Building Protein Systems in Explicit Solvent for more examples. **AmberTools** utilities from **AMBER MD package** contain a tool able to analyse **PDB files** and clean them for further usage, especially with the **AmberTools LEaP program**: the **pdb4amber tool**. The next step of the workflow is running this tool to analyse our **input PDB structure**.

For the particular **T4 Lysosyme** example, the most important property that is identified by the **pdb4amber** utility is the presence of **disulfide bridges** in the structure. Those are marked changing the residue names **from CYS to CYX**, which is the code that **AMBER force fields** use to distinguish between cysteines forming or not forming **disulfide bridges**. This will be used in the following step to correctly form a **bond** between these cysteine residues.

We invite you to check what the tool does with different, more complex structures (e.g. PDB code 6N3V).

Building Blocks used:

- pdb4amber_run from **biobb_amber.pdb4amber.pdb4amber_run**

```
# Import module
from biobb_amber.pdb4amber.pdb4amber_run import pdb4amber_run

# Create prop dict and inputs/outputs
output_pdb4amber_path = 'structure.pdb4amber.pdb'

# Create and launch bb
pdb4amber_run(input_pdb_path=nobme_pdb,
              output_pdb_path=output_pdb4amber_path,
              properties=prop)
```

1.3.8 Create ligand system topology

Building AMBER topology corresponding to the ligand structure. Force field used in this tutorial step is **amberGAFF**: General AMBER Force Field, designed for rational drug design.

- *Step 1*: Extract **ligand structure**.
- *Step 2*: Add **hydrogen atoms** if missing.
- *Step 3*: **Energetically minimize the system** with the new hydrogen atoms.
- *Step 4*: Generate **ligand topology** (parameters).

Building Blocks used:

- ExtractHeteroAtoms from **biobb_structure_utils.utils.extract_heteroatoms**

- ReduceAddHydrogens from **biobb_chemistry.ambertools.reduce_add_hydrogens**
- BabelMinimize from **biobb_chemistry.babelm.babel_minimize**
- AcpypeParamsAC from **biobb_chemistry.acpype.acpype_params_ac**

Step 1: Extract Ligand structure

```
# Create Ligand system topology, STEP 1
# Extracting Ligand JZ4
# Import module
from biobb_structure_utils.utils.extract_heteroatoms import extract_heteroatoms

# Create properties dict and inputs/outputs
ligandFile = ligandCode+'.pdb'

prop = {
     'heteroatoms' : [{"name": "JZ4"}]
}

extract_heteroatoms(input_structure_path=output_pdb4amber_path,
     output_heteroatom_path=ligandFile,
     properties=prop)
```

Step 2: Add hydrogen atoms

```
# Create Ligand system topology, STEP 2
# Reduce_add_hydrogens: add Hydrogen atoms to a small molecule (using Reduce tool from␣
↪Ambertools package)
# Import module
from biobb_chemistry.ambertools.reduce_add_hydrogens import reduce_add_hydrogens

# Create prop dict and inputs/outputs
output_reduce_h = ligandCode+'.reduce.H.pdb'

prop = {
    'nuclear' : 'true'
}

# Create and launch bb
reduce_add_hydrogens(input_path=ligandFile,
                     output_path=output_reduce_h,
                     properties=prop)
```

Step 3: Energetically minimize the system with the new hydrogen atoms.

```
# Create Ligand system topology, STEP 3
# Babel_minimize: Structure energy minimization of a small molecule after being modified␣
↪adding hydrogen atoms
# Import module
from biobb_chemistry.babelm.babel_minimize import babel_minimize

# Create prop dict and inputs/outputs
output_babel_min = ligandCode+'.H.min.mol2'

prop = {
    'method' : 'sd',
    'criteria' : '1e-10',
    'force_field' : 'GAFF'
}


# Create and launch bb
babel_minimize(input_path=output_reduce_h,
               output_path=output_babel_min,
               properties=prop)
```

Visualizing 3D structures

Visualizing the small molecule generated **PDB structures** using **NGL**:

- **Original Ligand Structure** (Left)
- **Ligand Structure with hydrogen atoms added** (with Reduce program) (Center)
- **Ligand Structure with hydrogen atoms added** (with Reduce program), **energy minimized** (with Open Babel) (Right)

```
# Show different structures generated (for comparison)

view1 = nglview.show_structure_file(ligandFile)
view1.add_representation(repr_type='ball+stick')
view1._remote_call('setSize', target='Widget', args=['350px','400px'])
view1.camera='orthographic'
view1
view2 = nglview.show_structure_file(output_reduce_h)
view2.add_representation(repr_type='ball+stick')
view2._remote_call('setSize', target='Widget', args=['350px','400px'])
view2.camera='orthographic'
view2
view3 = nglview.show_structure_file(output_babel_min)
view3.add_representation(repr_type='ball+stick')
view3._remote_call('setSize', target='Widget', args=['350px','400px'])
view3.camera='orthographic'
view3
ipywidgets.HBox([view1, view2, view3])
```

Step 4: Generate **ligand topology** (parameters).

```
# Create Ligand system topology, STEP 4
# Acpype_params_gmx: Generation of topologies for AMBER with ACPype
# Import module
from biobb_chemistry.acpype.acpype_params_ac import acpype_params_ac

# Create prop dict and inputs/outputs
output_acpype_inpcrd = ligandCode+'params.inpcrd'
output_acpype_frcmod = ligandCode+'params.frcmod'
output_acpype_lib = ligandCode+'params.lib'
output_acpype_prmtop = ligandCode+'params.prmtop'
output_acpype = ligandCode+'params'

prop = {
    'basename' : output_acpype,
    'charge' : mol_charge
}

# Create and launch bb
acpype_params_ac(input_path=output_babel_min,
                 output_path_inpcrd=output_acpype_inpcrd,
                 output_path_frcmod=output_acpype_frcmod,
                 output_path_lib=output_acpype_lib,
                 output_path_prmtop=output_acpype_prmtop,
                 properties=prop)
```

1.3.9 Create protein-ligand complex system topology

Building AMBER topology corresponding to the protein-ligand complex structure.

IMPORTANT: the previous pdb4amber building block is changing the proper cysteines residue naming in the PDB file from CYS to CYX so that this step can automatically identify and add the disulfide bonds to the system topology.

The **force field** used in this tutorial is **ff14SB** for the **protein**, an evolution of the **ff99SB** force field with improved accuracy of protein side chains and backbone parameters; and the **gaff** force field for the small molecule. **Water** molecules type used in this tutorial is **tip3p**. Adding **side chain atoms** and **hydrogen atoms** if missing. Forming **disulfide bridges** according to the info added in the previous step.

*NOTE: From this point on, the **protein-ligand complex structure and topology** generated can be used in a regular MD setup.*

Generating three output files:

- **AMBER structure** (PDB file)
- **AMBER topology** (AMBER Parmtop file)
- **AMBER coordinates** (AMBER Coordinate/Restart file)

Building Blocks used:

- leap_gen_top from **biobb_amber.leap.leap_gen_top**

```
# Import module
from biobb_amber.leap.leap_gen_top import leap_gen_top

# Create prop dict and inputs/outputs
output_pdb_path = 'structure.leap.pdb'
output_top_path = 'structure.leap.top'
output_crd_path = 'structure.leap.crd'

prop = {
    "forcefield" : ["protein.ff14SB","gaff"]
}

# Create and launch bb
leap_gen_top(input_pdb_path=output_pdb4amber_path,
            input_lib_path=output_acpype_lib,
            input_frcmod_path=output_acpype_frcmod,
            output_pdb_path=output_pdb_path,
            output_top_path=output_top_path,
            output_crd_path=output_crd_path,
            properties=prop)
```

Visualizing 3D structure

Visualizing the **PDB structure** using **NGL**. Try to identify the differences between the structure generated for the **system topology** and the **original one** (e.g. hydrogen atoms).

```
import nglview
import ipywidgets

# Show protein
view = nglview.show_structure_file(output_pdb_path)
view.clear_representations()
view.add_representation(repr_type='cartoon', selection='protein', opacity='0.4')
view.add_representation(repr_type='ball+stick', selection='protein')
view.add_representation(repr_type='ball+stick', radius='0.5', selection='JZ4')
view._remote_call('setSize', target='Widget', args=['','600px'])
view
```

1.3.10 Energetically minimize the structure

Energetically minimize the **protein-ligand complex structure** (in vacuo) using the **sander tool** from the **AMBER MD package**. This step is **relaxing the structure**, usually **constrained**, especially when coming from an X-ray **crystal structure**.

The **miminization process** is done in two steps:

- *Step 1*: **Hydrogen** minimization, applying **position restraints** (50 Kcal/mol.$Å^{2}$) to the **protein heavy atoms**.
- *Step 2*: **System** minimization, with **no restraints**.

Building Blocks used:

- sander_mdrun from **biobb_amber.sander.sander_mdrun**
- process_minout from **biobb_amber.process.process_minout**

Step 1: Minimize Hydrogens

Hydrogen minimization, applying **position restraints** (50 Kcal/mol.$Å^{2}$) to the **protein heavy atoms**.

```
# Import module
from biobb_amber.sander.sander_mdrun import sander_mdrun

# Create prop dict and inputs/outputs
output_h_min_traj_path = 'sander.h_min.x'
output_h_min_rst_path = 'sander.h_min.rst'
output_h_min_log_path = 'sander.h_min.log'

prop = {
    'simulation_type' : "min_vacuo",
    "mdin" : {
        'maxcyc' : 500,
        'ntpr' : 5,
        'ntr' : 1,
        'restraintmask' : '\":*&!@H=\"',
        'restraint_wt' : 50.0
    }
}

# Create and launch bb
sander_mdrun(input_top_path=output_top_path,
             input_crd_path=output_crd_path,
             input_ref_path=output_crd_path,
             output_traj_path=output_h_min_traj_path,
             output_rst_path=output_h_min_rst_path,
             output_log_path=output_h_min_log_path,
             properties=prop)
```

Checking Energy Minimization results

Checking **energy minimization** results. Plotting **potential energy** along time during the **minimization process**.

```
# Import module
from biobb_amber.process.process_minout import process_minout

# Create prop dict and inputs/outputs
output_h_min_dat_path = 'sander.h_min.energy.dat'

prop = {
    "terms" : ['ENERGY']
}

# Create and launch bb
process_minout(input_log_path=output_h_min_log_path,
            output_dat_path=output_h_min_dat_path,
            properties=prop)
```

```
#Read data from file and filter energy values higher than 1000 Kj/mol^-1
with open(output_h_min_dat_path,'r') as energy_file:
    x,y = map(
        list,
        zip(*[
            (float(line.split()[0]),float(line.split()[1]))
            for line in energy_file
            if not line.startswith(("#","@"))
            if float(line.split()[1]) < 1000
        ])
    )

plotly.offline.init_notebook_mode(connected=True)

fig = {
    "data": [go.Scatter(x=x, y=y)],
    "layout": go.Layout(title="Energy Minimization",
                        xaxis=dict(title = "Energy Minimization Step"),
                        yaxis=dict(title = "Potential Energy kcal/mol")
                       )
}

plotly.offline.iplot(fig)
```

Step 2: Minimize the system

System minimization, with **restraints** only on the **small molecule**, to avoid a possible change in position due to **protein repulsion**.

```
# Import module
from biobb_amber.sander.sander_mdrun import sander_mdrun

# Create prop dict and inputs/outputs
output_n_min_traj_path = 'sander.n_min.x'
output_n_min_rst_path = 'sander.n_min.rst'
output_n_min_log_path = 'sander.n_min.log'

prop = {
    'simulation_type' : "min_vacuo",
    "mdin" : {
        'maxcyc' : 500,
        'ntpr' : 5,
        'restraintmask' : '\":' + ligandCode + '\"',
        'restraint_wt' : 500.0
    }
}

# Create and launch bb
sander_mdrun(input_top_path=output_top_path,
             input_crd_path=output_h_min_rst_path,
             output_traj_path=output_n_min_traj_path,
             output_rst_path=output_n_min_rst_path,
             output_log_path=output_n_min_log_path,
             properties=prop)
```

Checking Energy Minimization results

Checking **energy minimization** results. Plotting **potential energy** by time during the **minimization process**.

```
# Import module
from biobb_amber.process.process_minout import process_minout

# Create prop dict and inputs/outputs
output_n_min_dat_path = 'sander.n_min.energy.dat'

prop = {
    "terms" : ['ENERGY']
}

# Create and launch bb
process_minout(input_log_path=output_n_min_log_path,
             output_dat_path=output_n_min_dat_path,
             properties=prop)
```

```
#Read data from file and filter energy values higher than 1000 Kj/mol^-1
with open(output_n_min_dat_path,'r') as energy_file:
```

(continues on next page)

(continued from previous page)

```
    x,y = map(
        list,
        zip(*[
            (float(line.split()[0]),float(line.split()[1]))
            for line in energy_file
            if not line.startswith(("#","@"))
            if float(line.split()[1]) < 1000
        ])
    )

plotly.offline.init_notebook_mode(connected=True)

fig = {
    "data": [go.Scatter(x=x, y=y)],
    "layout": go.Layout(title="Energy Minimization",
                        xaxis=dict(title = "Energy Minimization Step"),
                        yaxis=dict(title = "Potential Energy kcal/mol")
                       )
}

plotly.offline.iplot(fig)
```

1.3.11 Create solvent box and solvating the system

Define the unit cell for the **protein structure MD system** to fill it with water molecules. A **truncated octahedron box** is used to define the unit cell, with a **distance from the protein to the box edge of 9.0 Angstroms**. The solvent type used is the default **TIP3P** water model, a generic 3-point solvent model.

Building Blocks used:

- amber_to_pdb from **biobb_amber.ambpdb.amber_to_pdb**
- leap_solvate from **biobb_amber.leap.leap_solvate**

Getting minimized structure

Getting the result of the **energetic minimization** and converting it to **PDB format** to be then used as input for the **water box generation**. This is achieved by converting from **AMBER topology + coordinates** files to a **PDB file** using the **ambpdb** tool from the **AMBER MD package**.

```
# Import module
from biobb_amber.ambpdb.amber_to_pdb import amber_to_pdb
```

(continues on next page)

(continued from previous page)

```
# Create prop dict and inputs/outputs
output_ambpdb_path = 'structure.ambpdb.pdb'

# Create and launch bb
amber_to_pdb(input_top_path=output_top_path,
             input_crd_path=output_h_min_rst_path,
             output_pdb_path=output_ambpdb_path)
```

Create water box

Define the **unit cell** for the **protein-ligand complex structure MD system** and fill it with **water molecules**.

```
# Import module
from biobb_amber.leap.leap_solvate import leap_solvate

# Create prop dict and inputs/outputs
output_solv_pdb_path = 'structure.solv.pdb'
output_solv_top_path = 'structure.solv.parmtop'
output_solv_crd_path = 'structure.solv.crd'

prop = {
    "forcefield" : ["protein.ff14SB","gaff"],
    "water_type": "TIP3PBOX",
    "distance_to_molecule": "9.0",
    "box_type": "truncated_octahedron"
}

# Create and launch bb
leap_solvate(input_pdb_path=output_ambpdb_path,
             input_lib_path=output_acpype_lib,
             input_frcmod_path=output_acpype_frcmod,
             output_pdb_path=output_solv_pdb_path,
             output_top_path=output_solv_top_path,
             output_crd_path=output_solv_crd_path,
             properties=prop)
```

1.3.12 Adding ions

Neutralizing the system and adding an additional **ionic concentration** using the **leap tool** from the **AMBER MD package**. Using **Sodium (Na+)** and **Chloride (Cl-)** counterions and an **additional ionic concentration** of 150mM.

Building Blocks used:

- leap_add_ions from **biobb_amber.leap.leap_add_ions**

```
# Import module
from biobb_amber.leap.leap_add_ions import leap_add_ions

# Create prop dict and inputs/outputs
output_ions_pdb_path = 'structure.ions.pdb'
output_ions_top_path = 'structure.ions.parmtop'
output_ions_crd_path = 'structure.ions.crd'

prop = {
    "forcefield" : ["protein.ff14SB","gaff"],
    "neutralise" : True,
    "positive_ions_type": "Na+",
    "negative_ions_type": "Cl-",
    "ionic_concentration" : 150, # 150mM
    "box_type": "truncated_octahedron"
}

# Create and launch bb
leap_add_ions(input_pdb_path=output_solv_pdb_path,
             input_lib_path=output_acpype_lib,
             input_frcmod_path=output_acpype_frcmod,
            output_pdb_path=output_ions_pdb_path,
            output_top_path=output_ions_top_path,
            output_crd_path=output_ions_crd_path,
            properties=prop)
```

Visualizing 3D structure

Visualizing the **protein-ligand complex system** with the newly added **solvent box** and **counterions** using **NGL**. Note the **truncated octahedron box** filled with **water molecules** surrounding the **protein structure**, as well as the randomly placed **positive** (Na+, blue) and **negative** (Cl-, gray) **counterions**.

```
# Show protein
view = nglview.show_structure_file(output_ions_pdb_path)
view.clear_representations()
view.add_representation(repr_type='cartoon', selection='protein')
view.add_representation(repr_type='ball+stick', selection='solvent')
view.add_representation(repr_type='spacefill', selection='Cl- Na+')
view._remote_call('setSize', target='Widget', args=['','600px'])
view
```

1.3.13 Energetically minimize the system

Energetically minimize the **system** (protein structure + ligand + solvent + ions) using the **sander tool** from the **AMBER MD package**. **Restraining heavy atoms** with a force constant of 15 15 Kcal/mol.$Å^{2}$ to their initial positions.

- *Step 1*: Energetically minimize the **system** through 500 minimization cycles.
- *Step 2*: Checking **energy minimization** results. Plotting energy by time during the **minimization** process.

Building Blocks used:

- sander_mdrun from **biobb_amber.sander.sander_mdrun**
- process_minout from **biobb_amber.process.process_minout**

Step 1: Running Energy Minimization

The **minimization** type of the **simulation_type property** contains the main default parameters to run an **energy minimization**:

- imin = 1 ; Minimization flag, perform an energy minimization.
- maxcyc = 500; The maximum number of cycles of minimization.
- ntb = 1; Periodic boundaries: constant volume.
- ntmin = 2; Minimization method: steepest descent.

In this particular example, the method used to run the **energy minimization** is the default **steepest descent**, with a **maximum number of 500 cycles** and **periodic conditions**.

```
# Import module
from biobb_amber.sander.sander_mdrun import sander_mdrun

# Create prop dict and inputs/outputs
output_min_traj_path = 'sander.min.x'
output_min_rst_path = 'sander.min.rst'
output_min_log_path = 'sander.min.log'

prop = {
    "simulation_type" : "minimization",
    "mdin" : {
        'maxcyc' : 300, # Reducing the number of minimization steps for the sake of time
        'ntr' : 1,      # Overwritting restrain parameter
        'restraintmask' : '\"!:WAT,Cl-,Na+\"',      # Restraining solute
        'restraint_wt' : 15.0                       # With a force constant of 50 Kcal/
↪mol*A2
    }
}

# Create and launch bb
sander_mdrun(input_top_path=output_ions_top_path,
```

(continues on next page)

(continued from previous page)

```
            input_crd_path=output_ions_crd_path,
            input_ref_path=output_ions_crd_path,
            output_traj_path=output_min_traj_path,
            output_rst_path=output_min_rst_path,
            output_log_path=output_min_log_path,
            properties=prop)
```

Step 2: Checking Energy Minimization results

Checking **energy minimization** results. Plotting **potential energy** along time during the **minimization process**.

```
# Import module
from biobb_amber.process.process_minout import process_minout

# Create prop dict and inputs/outputs
output_dat_path = 'sander.min.energy.dat'

prop = {
    "terms" : ['ENERGY']
}

# Create and launch bb
process_minout(input_log_path=output_min_log_path,
            output_dat_path=output_dat_path,
            properties=prop)
```

```
#Read data from file and filter energy values higher than 1000 Kj/mol^-1
with open(output_dat_path,'r') as energy_file:
    x,y = map(
        list,
        zip(*[
            (float(line.split()[0]),float(line.split()[1]))
            for line in energy_file
            if not line.startswith(("#","@"))
            if float(line.split()[1]) < 1000
        ])
    )

plotly.offline.init_notebook_mode(connected=True)

fig = {
    "data": [go.Scatter(x=x, y=y)],
    "layout": go.Layout(title="Energy Minimization",
                        xaxis=dict(title = "Energy Minimization Step"),
                        yaxis=dict(title = "Potential Energy kcal/mol")
                       )
}

plotly.offline.iplot(fig)
```

1.3.14 Heating the system

Warming up the **prepared system** using the **sander tool** from the **AMBER MD package**. Going from 0 to the desired **temperature**, in this particular example, 300K. **Solute atoms restrained** (force constant of 10 Kcal/mol). Length 5ps.

- *Step 1*: Warming up the **system** through 500 MD steps.
- *Step 2*: Checking results for the **system warming up**. Plotting **temperature** along time during the **heating** process.

Building Blocks used:

- sander_mdrun from **biobb_amber.sander.sander_mdrun**
- process_mdout from **biobb_amber.process.process_mdout**

Step 1: Warming up the system

The **heat** type of the **simulation_type property** contains the main default parameters to run a **system warming up**:

- imin = 0; Run MD (no minimization)
- ntx = 5; Read initial coords and vels from restart file
- cut = 10.0; Cutoff for non bonded interactions in Angstroms
- ntr = 0; No restrained atoms
- ntc = 2; SHAKE for constraining length of bonds involving Hydrogen atoms
- ntf = 2; Bond interactions involving H omitted
- ntt = 3; Constant temperature using Langevin dynamics
- ig = -1; Seed for pseudo-random number generator
- ioutfm = 1; Write trajectory in netcdf format
- iwrap = 1; Wrap coords into primary box
- nstlim = 5000; Number of MD steps
- dt = 0.002; Time step (in ps)
- tempi = 0.0; Initial temperature (0 K)
- temp0 = 300.0; Final temperature (300 K)
- irest = 0; No restart from previous simulation
- ntb = 1; Periodic boundary conditions at constant volume
- gamma_ln = 1.0; Collision frequency for Langevin dynamics (in 1/ps)

In this particular example, the **heating** of the system is done in **2500 steps** (5ps) and is going **from 0K to 300K** (note that the number of steps has been reduced in this tutorial, for the sake of time).

```
# Import module
from biobb_amber.sander.sander_mdrun import sander_mdrun

# Create prop dict and inputs/outputs
output_heat_traj_path = 'sander.heat.netcdf'
output_heat_rst_path = 'sander.heat.rst'
output_heat_log_path = 'sander.heat.log'

prop = {
    "simulation_type" : "heat",
    "mdin" : {
        'nstlim' : 2500, # Reducing the number of steps for the sake of time (5ps)
        'ntr' : 1,     # Overwritting restrain parameter
        'restraintmask' : '\"!:WAT,Cl-,Na+\"',      # Restraining solute
        'restraint_wt' : 10.0                       # With a force constant of 10 Kcal/
↪mol*A2
    }
}

# Create and launch bb
sander_mdrun(input_top_path=output_ions_top_path,
             input_crd_path=output_min_rst_path,
             input_ref_path=output_min_rst_path,
             output_traj_path=output_heat_traj_path,
             output_rst_path=output_heat_rst_path,
             output_log_path=output_heat_log_path,
             properties=prop)
```

Step 2: Checking results from the system warming up

Checking **system warming up** output. Plotting **temperature** along time during the **heating process**.

```
# Import module
from biobb_amber.process.process_mdout import process_mdout

# Create prop dict and inputs/outputs
output_dat_heat_path = 'sander.md.temp.dat'

prop = {
    "terms" : ['TEMP']
}

# Create and launch bb
process_mdout(input_log_path=output_heat_log_path,
             output_dat_path=output_dat_heat_path,
             properties=prop)
```

```
#Read data from file and filter energy values higher than 1000 Kj/mol^-1
with open(output_dat_heat_path,'r') as energy_file:
    x,y = map(
        list,
        zip(*[
            (float(line.split()[0]),float(line.split()[1]))
            for line in energy_file
            if not line.startswith(("#","@"))
            if float(line.split()[1]) < 1000
        ])
    )

plotly.offline.init_notebook_mode(connected=True)

fig = {
    "data": [go.Scatter(x=x, y=y)],
    "layout": go.Layout(title="Heating process",
                        xaxis=dict(title = "Heating Step (ps)"),
                        yaxis=dict(title = "Temperature (K)")
                       )
}

plotly.offline.iplot(fig)
```

1.3.15 Equilibrate the system (NVT)

Equilibrate the **protein-ligand complex system** in **NVT ensemble** (constant Number of particles, Volume and Temperature). Protein **heavy atoms** will be restrained using position restraining forces: movement is permitted, but only after overcoming a substantial energy penalty. The utility of position restraints is that they allow us to equilibrate our solvent around our protein, without the added variable of structural changes in the protein.

- *Step 1*: Equilibrate the **protein system** with **NVT** ensemble.
- *Step 2*: Checking **NVT Equilibration** results. Plotting **system temperature** by time during the **NVT equilibration** process.

Building Blocks used:

- sander_mdrun from **biobb_amber.sander.sander_mdrun**
- process_mdout from **biobb_amber.process.process_mdout**

Step 1: Equilibrating the system (NVT)

The **nvt** type of the **simulation_type property** contains the main default parameters to run a **system equilibration in NVT ensemble**:

- imin = 0; Run MD (no minimization)
- ntx = 5; Read initial coords and vels from restart file
- cut = 10.0; Cutoff for non bonded interactions in Angstroms
- ntr = 0; No restrained atoms
- ntc = 2; SHAKE for constraining length of bonds involving Hydrogen atoms
- ntf = 2; Bond interactions involving H omitted
- ntt = 3; Constant temperature using Langevin dynamics
- ig = -1; Seed for pseudo-random number generator
- ioutfm = 1; Write trajectory in netcdf format
- iwrap = 1; Wrap coords into primary box
- nstlim = 5000; Number of MD steps
- dt = 0.002; Time step (in ps)
- irest = 1; Restart previous simulation
- ntb = 1; Periodic boundary conditions at constant volume
- gamma_ln = 5.0; Collision frequency for Langevin dynamics (in 1/ps)

In this particular example, the **NVT equilibration** of the system is done in **500 steps** (note that the number of steps has been reduced in this tutorial, for the sake of time).

```
# Import module
from biobb_amber.sander.sander_mdrun import sander_mdrun

# Create prop dict and inputs/outputs
output_nvt_traj_path = 'sander.nvt.netcdf'
output_nvt_rst_path = 'sander.nvt.rst'
output_nvt_log_path = 'sander.nvt.log'

prop = {
    "simulation_type" : 'nvt',
    "mdin" : {
        'nstlim' : 500, # Reducing the number of steps for the sake of time (1ps)
        'ntr' : 1,     # Overwritting restrain parameter
        'restraintmask' : '\"!:WAT,Cl-,Na+ & !@H=\"',      # Restraining solute heavy␣
↪atoms
        'restraint_wt' : 5.0                               # With a force constant of 5␣
↪Kcal/mol*A2
    }
}

# Create and launch bb
sander_mdrun(input_top_path=output_ions_top_path,
             input_crd_path=output_heat_rst_path,
```

(continues on next page)

(continued from previous page)

```
            input_ref_path=output_heat_rst_path,
            output_traj_path=output_nvt_traj_path,
            output_rst_path=output_nvt_rst_path,
            output_log_path=output_nvt_log_path,
            properties=prop)
```

Step 2: Checking NVT Equilibration results

Checking **NVT Equilibration** results. Plotting **system temperature** by time during the NVT equilibration process.

```
# Import module
from biobb_amber.process.process_mdout import process_mdout

# Create prop dict and inputs/outputs
output_dat_nvt_path = 'sander.md.nvt.temp.dat'

prop = {
    "terms" : ['TEMP']
}

# Create and launch bb
process_mdout(input_log_path=output_nvt_log_path,
            output_dat_path=output_dat_nvt_path,
            properties=prop)
```

```
#Read data from file and filter energy values higher than 1000 Kj/mol^-1
with open(output_dat_nvt_path,'r') as energy_file:
    x,y = map(
        list,
        zip(*[
            (float(line.split()[0]),float(line.split()[1]))
            for line in energy_file
            if not line.startswith(("#","@"))
            if float(line.split()[1]) < 1000
        ])
    )

plotly.offline.init_notebook_mode(connected=True)

fig = {
    "data": [go.Scatter(x=x, y=y)],
    "layout": go.Layout(title="NVT equilibration",
                        xaxis=dict(title = "Equilibration Step (ps)"),
                        yaxis=dict(title = "Temperature (K)")
                       )
}

plotly.offline.iplot(fig)
```

1.3.16 Equilibrate the system (NPT)

Equilibrate the **protein-ligand complex system** in **NPT ensemble** (constant Number of particles, Pressure and Temperature). Protein **heavy atoms** will be restrained using position restraining forces: movement is permitted, but only after overcoming a substantial energy penalty. The utility of position restraints is that they allow us to equilibrate our solvent around our protein, without the added variable of structural changes in the protein.

- *Step 1*: Equilibrate the **protein system** with **NPT** ensemble.
- *Step 2*: Checking **NPT Equilibration** results. Plotting **system pressure and density** by time during the **NVT equilibration** process.

Building Blocks used:

- sander_mdrun from **biobb_amber.sander.sander_mdrun**
- process_mdout from **biobb_amber.process.process_mdout**

Step 1: Equilibrating the system (NPT)

The **npt** type of the **simulation_type property** contains the main default parameters to run a **system equilibration in NPT ensemble**:

- imin = 0; Run MD (no minimization)
- ntx = 5; Read initial coords and vels from restart file
- cut = 10.0; Cutoff for non bonded interactions in Angstroms
- ntr = 0; No restrained atoms
- ntc = 2; SHAKE for constraining length of bonds involving Hydrogen atoms
- ntf = 2; Bond interactions involving H omitted
- ntt = 3; Constant temperature using Langevin dynamics
- ig = -1; Seed for pseudo-random number generator
- ioutfm = 1; Write trajectory in netcdf format
- iwrap = 1; Wrap coords into primary box
- nstlim = 5000; Number of MD steps
- dt = 0.002; Time step (in ps)
- irest = 1; Restart previous simulation
- gamma_ln = 5.0; Collision frequency for Langevin dynamics (in 1/ps)
- pres0 = 1.0; Reference pressure
- ntp = 1; Constant pressure dynamics: md with isotropic position scaling

- taup = 2.0; Pressure relaxation time (in ps)

In this particular example, the **NPT equilibration** of the system is done in **500 steps** (note that the number of steps has been reduced in this tutorial, for the sake of time).

```
# Import module
from biobb_amber.sander.sander_mdrun import sander_mdrun

# Create prop dict and inputs/outputs
output_npt_traj_path = 'sander.npt.netcdf'
output_npt_rst_path = 'sander.npt.rst'
output_npt_log_path = 'sander.npt.log'

prop = {
    "simulation_type" : 'npt',
    "mdin" : {
        'nstlim' : 500, # Reducing the number of steps for the sake of time (1ps)
        'ntr' : 1,     # Overwritting restrain parameter
        'restraintmask' : '\"!:WAT,Cl-,Na+ & !@H=\"',      # Restraining solute heavy␣
↪atoms
        'restraint_wt' : 2.5                               # With a force constant of 2.
↪5 Kcal/mol*A2
    }
}

# Create and launch bb
sander_mdrun(input_top_path=output_ions_top_path,
             input_crd_path=output_nvt_rst_path,
             input_ref_path=output_nvt_rst_path,
             output_traj_path=output_npt_traj_path,
             output_rst_path=output_npt_rst_path,
             output_log_path=output_npt_log_path,
             properties=prop)
```

Step 2: Checking NPT Equilibration results

Checking **NPT Equilibration** results. Plotting **system pressure and density** by time during the **NPT equilibration** process.

```
# Import module
from biobb_amber.process.process_mdout import process_mdout

# Create prop dict and inputs/outputs
output_dat_npt_path = 'sander.md.npt.dat'

prop = {
    "terms" : ['PRES','DENSITY']
}

# Create and launch bb
process_mdout(input_log_path=output_npt_log_path,
```

(continues on next page)

(continued from previous page)

```
            output_dat_path=output_dat_npt_path,
            properties=prop)
```

```
# Read pressure and density data from file
with open(output_dat_npt_path,'r') as pd_file:
    x,y,z = map(
        list,
        zip(*[
            (float(line.split()[0]),float(line.split()[1]),float(line.split()[2]))
            for line in pd_file
            if not line.startswith(("#","@"))
        ])
    )

plotly.offline.init_notebook_mode(connected=True)

trace1 = go.Scatter(
    x=x,y=y
)
trace2 = go.Scatter(
    x=x,y=z
)

fig = subplots.make_subplots(rows=1, cols=2, print_grid=False)

fig.append_trace(trace1, 1, 1)
fig.append_trace(trace2, 1, 2)

fig['layout']['xaxis1'].update(title='Time (ps)')
fig['layout']['xaxis2'].update(title='Time (ps)')
fig['layout']['yaxis1'].update(title='Pressure (bar)')
fig['layout']['yaxis2'].update(title='Density (Kg*m^-3)')

fig['layout'].update(title='Pressure and Density during NPT Equilibration')
fig['layout'].update(showlegend=False)

plotly.offline.iplot(fig)
```

1.3.17 Free Molecular Dynamics Simulation

Upon completion of the **two equilibration phases (NVT and NPT)**, the system is now well-equilibrated at the desired temperature and pressure. The **position restraints** can now be released. The last step of the **protein** MD setup is a short, **free MD simulation**, to ensure the robustness of the system.

- *Step 1*: Run short MD simulation of the **protein system**.
- *Step 2*: Checking results for the final step of the setup process, the **free MD run**. Plotting **Root Mean Square deviation (RMSd)** and **Radius of Gyration (Rgyr)** by time during the **free MD run** step.

Building Blocks used:

- sander_mdrun from **biobb_amber.sander.sander_mdrun**
- process_mdout from **biobb_amber.process.process_mdout**
- cpptraj_rms from **biobb_analysis.cpptraj.cpptraj_rms**
- cpptraj_rgyr from **biobb_analysis.cpptraj.cpptraj_rgyr**

Step 1: Creating portable binary run file to run a free MD simulation

The **free** type of the **simulation_type property** contains the main default parameters to run an **unrestrained MD simulation**:

- imin = 0; Run MD (no minimization)
- ntx = 5; Read initial coords and vels from restart file
- cut = 10.0; Cutoff for non bonded interactions in Angstroms
- ntr = 0; No restrained atoms
- ntc = 2; SHAKE for constraining length of bonds involving Hydrogen atoms
- ntf = 2; Bond interactions involving H omitted
- ntt = 3; Constant temperature using Langevin dynamics
- ig = -1; Seed for pseudo-random number generator
- ioutfm = 1; Write trajectory in netcdf format
- iwrap = 1; Wrap coords into primary box
- nstlim = 5000; Number of MD steps
- dt = 0.002; Time step (in ps)

In this particular example, a short, **5ps-length** simulation (2500 steps) is run, for the sake of time.

```
# Import module
from biobb_amber.sander.sander_mdrun import sander_mdrun

# Create prop dict and inputs/outputs
output_free_traj_path = 'sander.free.netcdf'
output_free_rst_path = 'sander.free.rst'
```

(continues on next page)

(continued from previous page)

```
output_free_log_path = 'sander.free.log'

prop = {
    "simulation_type" : 'free',
    "mdin" : {
        'nstlim' : 2500, # Reducing the number of steps for the sake of time (5ps)
        'ntwx' : 500  # Print coords to trajectory every 500 steps (1 ps)
    }
}

# Create and launch bb
sander_mdrun(input_top_path=output_ions_top_path,
             input_crd_path=output_npt_rst_path,
             output_traj_path=output_free_traj_path,
             output_rst_path=output_free_rst_path,
             output_log_path=output_free_log_path,
             properties=prop)
```

Step 2: Checking free MD simulation results

Checking results for the final step of the setup process, the **free MD run**. Plotting **Root Mean Square deviation (RMSd)** and **Radius of Gyration (Rgyr)** by time during the **free MD run** step. **RMSd** against the **experimental structure** (input structure of the pipeline) and against the **minimized and equilibrated structure** (output structure of the NPT equilibration step).

```
# cpptraj_rms: Computing Root Mean Square deviation to analyse structural stability
#              RMSd against minimized and equilibrated snapshot (backbone atoms)

# Import module
from biobb_analysis.ambertools.cpptraj_rms import cpptraj_rms

# Create prop dict and inputs/outputs
output_rms_first = pdbCode+'_rms_first.dat'

prop = {
    'mask': 'backbone',
    'reference': 'first'
}

# Create and launch bb
cpptraj_rms(input_top_path=output_ions_top_path,
             input_traj_path=output_free_traj_path,
             output_cpptraj_path=output_rms_first,
             properties=prop)
```

```
# cpptraj_rms: Computing Root Mean Square deviation to analyse structural stability
#              RMSd against experimental structure (backbone atoms)

# Import module
```

(continues on next page)

(continued from previous page)

```
from biobb_analysis.ambertools.cpptraj_rms import cpptraj_rms

# Create prop dict and inputs/outputs
output_rms_exp = pdbCode+'_rms_exp.dat'

prop = {
    'mask': 'backbone',
    'reference': 'experimental'
}

# Create and launch bb
cpptraj_rms(input_top_path=output_ions_top_path,
            input_traj_path=output_free_traj_path,
            output_cpptraj_path=output_rms_exp,
            input_exp_path=output_pdb_path,
            properties=prop)
```

```
# Read RMS vs first snapshot data from file
with open(output_rms_first,'r') as rms_first_file:
    x,y = map(
        list,
        zip(*[
            (float(line.split()[0]),float(line.split()[1]))
            for line in rms_first_file
            if not line.startswith(("#","@"))
        ])
    )

# Read RMS vs experimental structure data from file
with open(output_rms_exp,'r') as rms_exp_file:
    x2,y2 = map(
        list,
        zip(*[
            (float(line.split()[0]),float(line.split()[1]))
            for line in rms_exp_file
            if not line.startswith(("#","@"))
        ])
    )

trace1 = go.Scatter(
    x = x,
    y = y,
    name = 'RMSd vs first'
)

trace2 = go.Scatter(
    x = x,
    y = y2,
    name = 'RMSd vs exp'
)

data = [trace1, trace2]
```

(continues on next page)

(continued from previous page)

```
plotly.offline.init_notebook_mode(connected=True)

fig = {
    "data": data,
    "layout": go.Layout(title="RMSd during free MD Simulation",
                        xaxis=dict(title = "Time (ps)"),
                        yaxis=dict(title = "RMSd (Angstrom)")
                       )
}

plotly.offline.iplot(fig)
```

```
# cpptraj_rgyr: Computing Radius of Gyration to measure the protein compactness during␣
↪the free MD simulation

# Import module
from biobb_analysis.ambertools.cpptraj_rgyr import cpptraj_rgyr

# Create prop dict and inputs/outputs
output_rgyr = pdbCode+'_rgyr.dat'

prop = {
    'mask': 'backbone'
}

# Create and launch bb
cpptraj_rgyr(input_top_path=output_ions_top_path,
             input_traj_path=output_free_traj_path,
             output_cpptraj_path=output_rgyr,
             properties=prop)
```

```
# Read Rgyr data from file
with open(output_rgyr,'r') as rgyr_file:
    x,y = map(
        list,
        zip(*[
            (float(line.split()[0]),float(line.split()[1]))
            for line in rgyr_file
            if not line.startswith(("#","@"))
        ])
    )

plotly.offline.init_notebook_mode(connected=True)

fig = {
    "data": [go.Scatter(x=x, y=y)],
    "layout": go.Layout(title="Radius of Gyration",
                        xaxis=dict(title = "Time (ps)"),
                        yaxis=dict(title = "Rgyr (Angstrom)")
```

(continues on next page)

(continued from previous page)

```
                        )
}

plotly.offline.iplot(fig)
```

1.3.18 Post-processing and Visualizing resulting 3D trajectory

Post-processing and Visualizing the **protein system** MD setup **resulting trajectory** using **NGL**

- *Step 1*: *Imaging* the resulting trajectory, **stripping out water molecules and ions** and **correcting periodicity issues**.
- *Step 2*: Visualizing the *imaged* trajectory using the *dry* structure as a **topology**.

Building Blocks used:

- cpptraj_image from **biobb_analysis.cpptraj.cpptraj_image**

Step 1: *Imaging* the resulting trajectory.

Stripping out **water molecules and ions** and **correcting periodicity issues**

```
# cpptraj_image: "Imaging" the resulting trajectory
#                Removing water molecules and ions from the resulting structure

# Import module
from biobb_analysis.ambertools.cpptraj_image import cpptraj_image

# Create prop dict and inputs/outputs
output_imaged_traj = pdbCode+'_imaged_traj.trr'

prop = {
    'mask': 'solute',
    'format': 'trr'
}

# Create and launch bb
cpptraj_image(input_top_path=output_ions_top_path,
            input_traj_path=output_free_traj_path,
            output_cpptraj_path=output_imaged_traj,
            properties=prop)
```

Step 2: Visualizing the generated dehydrated trajectory.

Using the **imaged trajectory** (output of the *Post-processing step 1*) with the **dry structure** (output of

```
# Show trajectory
view = nglview.show_simpletraj(nglview.SimpletrajTrajectory(output_imaged_traj, output_
↪ambpdb_path), gui=True)
view.clear_representations()
view.add_representation('cartoon', color='sstruc')
view.add_representation('licorice', selection='JZ4', color='element', radius=1)
view
```

1.3.19 Output files

Important **Output files** generated:

- structure.ions.pdb: **System structure** of the MD setup protocol. Structure generated during the MD setup and used in the MD simulation. With hydrogen atoms, solvent box and counterions.
- sander.free.netcdf: **Final trajectory** of the MD setup protocol.
- sander.free.rst: **Final checkpoint file**, with information about the state of the simulation. It can be used to **restart** or **continue** a MD simulation.
- structure.ions.parmtop: **Final topology** of the MD system in AMBER Parm7 format.

Analysis (MD setup check) output files generated:

- 3htb_rms_first.dat: **Root Mean Square deviation (RMSd)** against **minimized and equilibrated structure** of the final **free MD run step**.
- 3htb_rms_exp.dat: **Root Mean Square deviation (RMSd)** against **experimental structure** of the final **free MD run step**.
- 3htb_rgyr.dat: **Radius of Gyration** of the final **free MD run step** of the **setup pipeline**.

1.3.20 Questions & Comments

Questions, issues, suggestions and comments are really welcome!

- GitHub issues:
  - https://github.com/bioexcel/biobb
- BioExcel forum:
  - https://ask.bioexcel.eu/c/BioExcel-Building-Blocks-library

1.4 AMBER Constant pH MD setup tutorial using BioExcel Building Blocks (biobb)

1.4.1 Partly based on:

- AMBER Advanced Tutorial 18: Constant pH MD Example, Calculating pKas for titratable side chains in HEWL by Jason Swails and T. Dwight McGee Jr.
- AMBER Advanced Tutorial 33: Constant pH and Redox Potential MD Example: Predicting pH-dependent standard Redox Potential values by Vinícius Wilian D. Cruzeiro.
- Modeling of pH sensors for CLN025 beta-hairpin by Jordi Juárez, Barril Lab, University of Barcelona.
- Constant pH MD simulation tutorial of BPTI protein in implicit solvent by Wei Zhang, University of the Pacific Stockton.

This tutorial aims to illustrate the process of **setting up a simulation system** to run **constant pH** Molecular Dynamics simulations with **AMBER**, step by step, using the **BioExcel Building Blocks library (biobb)** wrapping the **AmberTools** utility from the **AMBER package**. The particular example used is the **Bovine Pancreatic Trypsin Inhibitor (BPTI)** protein (PDB code 6PTI).

1.4.2 Settings

Biobb modules used

- biobb_io: Tools to fetch biomolecular data from public databases.
- biobb_amber: Tools to setup and run Molecular Dynamics simulations with AmberTools.

Auxiliar libraries used

- nb_conda_kernels: Enables a Jupyter Notebook or JupyterLab application in one conda environment to access kernels for Python, R, and other languages found in other environments.
- nglview: Jupyter/IPython widget to interactively view molecular structures and trajectories in notebooks.
- ipywidgets: Interactive HTML widgets for Jupyter notebooks and the IPython kernel.
- plotly: Python interactive graphing library integrated in Jupyter notebooks.

Conda Installation and Launch

```
git clone https://github.com/bioexcel/biobb_wf_amber_md_setup.git
cd biobb_wf_amber_md_setup
conda env create -f conda_env/environment.yml
conda activate biobb_AMBER_MDsetup_tutorials
jupyter-nbextension enable --py --user widgetsnbextension
jupyter-nbextension enable --py --user nglview
jupyter-notebook biobb_wf_amber_md_setup/notebooks/mdsetup_ph/biobb_amber_CpHMD_notebook.
↪ipynb
```

1.4.3 Pipeline steps

1. *Input Parameters*
2. *Fetching the PDB structure*
3. *Preparing the PDB file for Amber*
4. *Create Protein System Topology*
5. *Create Solvent Box and Solvating the System*
6. *Adding Ions*
7. *Generating the constant pH input file*
8. *Energetically Minimize the System*
9. *Heating the System*
10. *Equilibrate the System (NVT)*
11. *Equilibrate the System (NPT)*
12. *Constant pH Molecular Dynamics Simulation*
13. *Output Files*
14. *Questions & Comments*

1.4.4 Input parameters

Input parameters needed:

- **pdbCode**: PDB code of the protein structure (e.g. 6PTI)

```
import nglview
import ipywidgets
import plotly
from plotly import subplots
import plotly.graph_objs as go

pdbCode="6PTI"
```

1.4.5 Fetching PDB structure

Downloading **PDB structure** with the **protein molecule** from the RCSB PDB database. Alternatively, a **PDB file** can be used as starting structure.

Building Blocks used:

- pdb from **biobb_io.api.pdb**

```
# Import module
from biobb_io.api.pdb import pdb

# Create properties dict and inputs/outputs
downloaded_pdb = pdbCode+'.pdb'

prop = {
    'pdb_code': pdbCode
}

#Create and launch bb
pdb(output_pdb_path=downloaded_pdb,
    properties=prop)
```

Visualizing 3D structure

Visualizing the downloaded/given **PDB structure** using **NGL**.

```
# Show protein
view = nglview.show_structure_file(downloaded_pdb)
view.add_representation(repr_type='ball+stick', selection='all')
view._remote_call('setSize', target='Widget', args=['','600px'])
view
```

1.4.6 Preparing PDB file for AMBER

Before starting a **protein MD setup**, it is always strongly recommended to take a look at the initial structure and try to identify important **properties** and also possible **issues**. These properties and issues can be serious, as for example the definition of **disulfide bridges**, the presence of a **non-standard aminoacids** or **ligands**, or **missing residues**. Other **properties** and **issues** might not be so serious, but they still need to be addressed before starting the **MD setup process**. **Missing hydrogen atoms**, presence of **alternate atomic location indicators** or **inserted residue codes** (see PDB file format specification) are examples of these not so crucial characteristics. Please visit the AMBER tutorial: Building Protein Systems in Explicit Solvent for more examples. **AmberTools** utilities from **AMBER MD package** contain a tool able to analyse **PDB files** and clean them for further usage, especially with the **AmberTools LEaP program**: the **pdb4amber tool**. The next step of the workflow is running this tool to analyse our **input PDB structure**.

For the particular **BPTI** example, the most important features that are going to be used from the **pdb4amber** utility is the identification of **disulfide bridges** in the structure and the renaming of **ionizable residues** for our **constant pH** calculation. **Disulfide bridges** are marked changing the residue names **from CYS to CYX**, which is the code that **AMBER force fields** use to distinguish between cysteines forming or not forming **disulfide bridges**. This will be used in the following step to correctly form a **bond** between these cysteine residues. **Ionizable residues** are also marked changing the residue names, from the deprotonated form to the protonated one (e.g. **from ASN to AS4, from GLN to GL4, and from HIS to HIP**). **Cysteine, Lysine and Tyrosine** residue names are not changed as they are already in their protonated form at **phisiological pH** (pKa > 7.4).

Building Blocks used:

- pdb4amber_run from **biobb_amber.pdb4amber.pdb4amber_run**

```
# Import module
from biobb_amber.pdb4amber.pdb4amber_run import pdb4amber_run

# Create prop dict and inputs/outputs
output_pdb4amber_path = 'structure.pdb4amber.pdb'

prop = {
    'constant_pH' : True
}

# Create and launch bb
pdb4amber_run(input_pdb_path=downloaded_pdb,
              output_pdb_path=output_pdb4amber_path,
              properties=prop)
```

Visualizing 3D structure

Visualizing the **PDB structure** with modified residue names for **ionizable residues** using **NGL**. **GL4 and AS4** residues are highlighted.

```
# Show protein
view = nglview.show_structure_file(output_pdb4amber_path)
view.add_representation(repr_type='ball+stick', selection='all')
view.add_representation(repr_type='ball+stick', radius='0.5', selection='GL4 AS4')
view._remote_call('setSize', target='Widget', args=['','600px'])
view
```

1.4.7 Create protein system topology

Building AMBER topology corresponding to the **protein structure**.

The **force field** used in this tutorial is **ff14SB**, an evolution of the **ff99SB** force field with improved accuracy of protein side chains and backbone parameters, and the **constph** force field, including the **constant pH** parameters. **Water** molecules type used in this tutorial is **tip3p**.

Generating three output files:

- **AMBER structure** (PDB file)
- **AMBER topology** (AMBER Parmtop file)
- **AMBER coordinates** (AMBER Coordinate/Restart file)

Building Blocks used:

- leap_gen_top from **biobb_amber.leap.leap_gen_top**

```
# Import module
from biobb_amber.leap.leap_gen_top import leap_gen_top

# Create prop dict and inputs/outputs
output_pdb_path = 'structure.leap.pdb'
output_top_path = 'structure.leap.top'
output_crd_path = 'structure.leap.crd'

prop = {
    "forcefield" : ["protein.ff14SB","constph"]
}

# Create and launch bb
leap_gen_top(input_pdb_path=output_pdb4amber_path,
             output_pdb_path=output_pdb_path,
             output_top_path=output_top_path,
             output_crd_path=output_crd_path,
             properties=prop)
```

Visualizing 3D structure

Visualizing the **PDB structure** after topology generation using **NGL**. Note the newly added **hydrogen atoms**, in particular, the ones for the highlighted protonated **ionizable residues** (GL4, AS4).

```
# Show protein
view = nglview.show_structure_file(output_pdb_path)
view.add_representation(repr_type='ball+stick', selection='all')
view.add_representation(repr_type='ball+stick', radius='0.3', selection='GL4 AS4')
view._remote_call('setSize', target='Widget', args=['','600px'])
view
```

1.4.8 Create solvent box and solvating the system

Define the unit cell for the **protein structure MD system** to fill it with water molecules. A **truncated octahedron box** is used to define the unit cell, with a **distance from the structure to the box edge of 9.0 Angstroms**. The solvent type used is the default **TIP3P** water model, a generic 3-point solvent model.

Building Blocks used:

- leap_solvate from **biobb_amber.leap.leap_solvate**

```
# Import module
from biobb_amber.leap.leap_solvate import leap_solvate

# Create prop dict and inputs/outputs
output_solv_pdb_path = 'structure.solv.pdb'
output_solv_top_path = 'structure.solv.parmtop'
output_solv_crd_path = 'structure.solv.crd'

prop = {
    "forcefield" : ["protein.ff14SB","constph"],
    "water_type": "TIP3PBOX",
    "distance_to_molecule": "9.0",
    "box_type": "truncated_octahedron"
}

# Create and launch bb
leap_solvate(input_pdb_path=output_pdb_path,
            output_pdb_path=output_solv_pdb_path,
            output_top_path=output_solv_top_path,
            output_crd_path=output_solv_crd_path,
            properties=prop)
```

Visualizing 3D structure

Visualizing the solvated **PDB structure** using **NGL**.

```
# Show protein
view = nglview.show_structure_file(output_solv_pdb_path)
view.clear_representations()
view.add_representation(repr_type='cartoon', selection='protein')
view.add_representation(repr_type='ball+stick', selection='protein')
view.add_representation(repr_type='line', selection='solvent')
view._remote_call('setSize', target='Widget', args=['','600px'])
view
```

1.4.9 Adding ions

Neutralizing the system and adding an additional **ionic concentration** using the **leap tool** from the **AMBER MD package**. Using **Sodium (Na+)** and **Chloride (Cl-)** counterions and an **additional ionic concentration** of 150mM.

Building Blocks used:

- leap_add_ions from **biobb_amber.leap.leap_add_ions**

```
# Import module
from biobb_amber.leap.leap_add_ions import leap_add_ions

# Create prop dict and inputs/outputs
output_ions_pdb_path = 'structure.ions.pdb'
output_ions_top_path = 'structure.ions.parmtop'
output_ions_crd_path = 'structure.ions.crd'

prop = {
    "forcefield" : ["protein.ff14SB","constph"],
    "neutralise" : True,
    "box_type": "truncated_octahedron"
}

# Create and launch bb
leap_add_ions(input_pdb_path=output_solv_pdb_path,
            output_pdb_path=output_ions_pdb_path,
            output_top_path=output_ions_top_path,
            output_crd_path=output_ions_crd_path,
            properties=prop)
```

Visualizing 3D structure

Visualizing the solvated **PDB structure** with the newly added **counterions** (highlighted) using **NGL**.

```
# Show protein
view = nglview.show_structure_file(output_ions_pdb_path)
view.clear_representations()
view.add_representation(repr_type='cartoon', selection='protein')
view.add_representation(repr_type='ball+stick', selection='protein')
view.add_representation(repr_type='line', selection='solvent')
view.add_representation(repr_type='spacefill', selection='Cl- Na+', color='green')
view._remote_call('setSize', target='Widget', args=['','600px'])
view
```

1.4.10 Generating the constant pH input file

Generating the **constant pH** input file (***cpin file***) using the **cpinutil.py** program from the **AmberTools MD package**. This step is identifying which residues should be **titrated** during the course of the **MD simulation**. In this particular example, the residues we are interested in **titrating** during the simulation are **Glutamates (GL4)**, **Aspartates (AS4)**, **Cysteines (CYS)**, **Lysines (LYS)** and **Tyrosines (TYR)** (the structure used in this example doesn't contain Histidine residues).

Building Blocks used:

- parmed_cpinutil from **biobb_amber.parmed.parmed_cpinutil**

```
# Import module
from biobb_amber.parmed.parmed_cpinutil import parmed_cpinutil

# Create prop dict and inputs/outputs
output_cpin_path = 'structure.cpin'
output_top_cpin_path = 'structure.cpH.parmtop'

prop = {
    "igb" : 2,
    "resnames": "AS4 GL4 CYS LYS TYR", # No Histidines in our structure
    "system": "BPTI"
}

# Create and launch bb
parmed_cpinutil(input_top_path=output_ions_top_path,
           output_cpin_path=output_cpin_path,
           output_top_path=output_top_cpin_path,
           properties=prop)
```

1.4.11 Energetically minimize the system

Energetically minimize the **system** (protein structure + solvent + ions) using the **sander tool** from the **AMBER MD package**. **Restraining backbone atoms** with a force constant of 10 Kcal/mol.$Å^{2}$ to their initial positions.

- *Step 1*: Energetically minimize the **system** through 500 minimization cycles.
- *Step 2*: Checking **energy minimization** results. Plotting energy by time during the **minimization** process.

Building Blocks used:

- sander_mdrun from **biobb_amber.sander.sander_mdrun**
- process_minout from **biobb_amber.process.process_minout**

Step 1: Energetically minimize the system

System minimization, applying **position restraints** (10 Kcal/mol.$Å^{2}$) to the **protein backbone atoms**.

```
# Import module
from biobb_amber.sander.sander_mdrun import sander_mdrun

# Create prop dict and inputs/outputs
output_min_traj_path = 'sander.cpH.x'
output_min_rst_path = 'sander.cpH.rst'
output_min_log_path = 'sander.cpH.log'

prop = {
    "simulation_type" : "minimization",
    "mdin" : {
        'maxcyc' : 500,
        'ntr' : 1,           # Turn on positional restraints
        'restraint_wt' : 10,  # 10 kcal/mol/A**2 restraint force constant
        'restraintmask' : '\"@CA,C,O,N\"' # Restraints on the backbone atoms only
    }
}

# Create and launch bb
sander_mdrun(input_top_path=output_top_cpin_path,
             input_crd_path=output_ions_crd_path,
             input_ref_path=output_ions_crd_path,
             output_traj_path=output_min_traj_path,
             output_rst_path=output_min_rst_path,
             output_log_path=output_min_log_path,
             properties=prop)
```

Step 2: Checking Energy Minimization results

Checking **energy minimization** results. Plotting **potential energy** along time during the **minimization process**.

```
# Import module
from biobb_amber.process.process_minout import process_minout

# Create prop dict and inputs/outputs
output_h_min_dat_path = 'sander.min.energy.dat'

prop = {
    "terms" : ['ENERGY']
}

# Create and launch bb
process_minout(input_log_path=output_min_log_path,
             output_dat_path=output_h_min_dat_path,
             properties=prop)
```

```
# Read data from file and filter energy values higher than 1000 Kj/mol^-1
with open(output_h_min_dat_path,'r') as energy_file:
    x,y = map(
        list,
        zip(*[
            (float(line.split()[0]),float(line.split()[1]))
            for line in energy_file
            if not line.startswith(("#","@"))
            if float(line.split()[1]) < 1000
        ])
    )

plotly.offline.init_notebook_mode(connected=True)

fig = {
    "data": [go.Scatter(x=x, y=y)],
    "layout": go.Layout(title="Energy Minimization",
                        xaxis=dict(title = "Energy Minimization Step"),
                        yaxis=dict(title = "Potential Energy kcal/mol")
                       )
}

plotly.offline.iplot(fig)
```

1.4.12 Heating the system

Warming up the **prepared system** using the **sander tool** from the **AMBER MD package**. Going from 0 to the desired **temperature**, in this particular example, 300K. **Protein backbone atoms restrained** (force constant of 2 Kcal/mol). Length 5ps.

- *Step 1*: Warming up the **system** through 500 MD steps.
- *Step 2*: Checking results for the **system warming up**. Plotting **temperature** along time during the **heating** process.

Building Blocks used:

- sander_mdrun from **biobb_amber.sander.sander_mdrun**
- process_mdout from **biobb_amber.process.process_mdout**

Step 1: Warming up the system

The **heat** type of the **simulation_type property** contains the main default parameters to run a **system warming up**:

- imin = 0; Run MD (no minimization)
- ntx = 5; Read initial coords and vels from restart file
- cut = 10.0; Cutoff for non bonded interactions in Angstroms
- ntr = 0; No restrained atoms
- ntc = 2; SHAKE for constraining length of bonds involving Hydrogen atoms
- ntf = 2; Bond interactions involving H omitted
- ntt = 3; Constant temperature using Langevin dynamics
- ig = -1; Seed for pseudo-random number generator
- ioutfm = 1; Write trajectory in netcdf format
- iwrap = 1; Wrap coords into primary box
- nstlim = 5000; Number of MD steps
- dt = 0.002; Time step (in ps)
- tempi = 0.0; Initial temperature (0 K)
- temp0 = 300.0; Final temperature (300 K)
- irest = 0; No restart from previous simulation
- ntb = 1; Periodic boundary conditions at constant volume
- gamma_ln = 1.0; Collision frequency for Langevin dynamics (in 1/ps)

In this particular example, the **heating** of the system is done in **2500 steps** (5ps) and is going **from 0K to 300K** (note that the number of steps has been reduced in this tutorial, for the sake of time).

```
# Import module
from biobb_amber.sander.sander_mdrun import sander_mdrun

# Create prop dict and inputs/outputs
output_heat_traj_path = 'sander.heat.netcdf'
output_heat_rst_path = 'sander.heat.rst'
output_heat_log_path = 'sander.heat.log'

prop = {
    "simulation_type" : "heat",
    "mdin" : {
        'nstlim' : 2500,      # Reducing the number of steps for the sake of time (5ps)
        'ntr' : 1,            # Turn on positional restraints
        'restraintmask' : '\"@CA,C,O,N\"',         # Restraining protein backbone atoms
        'restraint_wt' : 2.0                       # With a force constant of 2 Kcal/
↪mol*A2
    }
}

# Create and launch bb
sander_mdrun(input_top_path=output_top_cpin_path,
```

(continues on next page)

(continued from previous page)

```
            input_crd_path=output_min_rst_path,
            input_ref_path=output_min_rst_path,
            output_traj_path=output_heat_traj_path,
            output_rst_path=output_heat_rst_path,
            output_log_path=output_heat_log_path,
            properties=prop)
```

Step 2: Checking results from the system warming up

Checking **system warming up** output. Plotting **temperature** along time during the **heating process**.

```
# Import module
from biobb_amber.process.process_mdout import process_mdout

# Create prop dict and inputs/outputs
output_dat_heat_path = 'sander.md.temp.dat'

prop = {
    "terms" : ['TEMP']
}

# Create and launch bb
process_mdout(input_log_path=output_heat_log_path,
            output_dat_path=output_dat_heat_path,
            properties=prop)
```

```
#Read data from file and filter energy values higher than 1000 Kj/mol^-1
with open(output_dat_heat_path,'r') as energy_file:
    x,y = map(
        list,
        zip(*[
            (float(line.split()[0]),float(line.split()[1]))
            for line in energy_file
            if not line.startswith(("#","@"))
            if float(line.split()[1]) < 1000
        ])
    )

plotly.offline.init_notebook_mode(connected=True)

fig = {
    "data": [go.Scatter(x=x, y=y)],
    "layout": go.Layout(title="Heating process",
                        xaxis=dict(title = "Heating Step (ps)"),
                        yaxis=dict(title = "Temperature (K)")
                       )
}

plotly.offline.iplot(fig)
```

1.4.13 Equilibrate the system (NVT)

Equilibrate the **protein system** in **NVT ensemble** (constant Number of particles, Volume and Temperature). Protein **backbone atoms** will be restrained using position restraining forces: movement is permitted, but only after overcoming a substantial energy penalty.

- *Step 1*: Equilibrate the **protein system** with **NVT** ensemble.
- *Step 2*: Checking **NVT Equilibration** results. Plotting **system temperature** by time during the **NVT equilibration** process.

Building Blocks used:

- sander_mdrun from **biobb_amber.sander.sander_mdrun**
- process_mdout from **biobb_amber.process.process_mdout**

Step 1: Equilibrating the system (NVT)

The **nvt** type of the **simulation_type property** contains the main default parameters to run a **system equilibration in NVT ensemble**:

- imin = 0; Run MD (no minimization)
- ntx = 5; Read initial coords and vels from restart file
- cut = 10.0; Cutoff for non bonded interactions in Angstroms
- ntr = 0; No restrained atoms
- ntc = 2; SHAKE for constraining length of bonds involving Hydrogen atoms
- ntf = 2; Bond interactions involving H omitted
- ntt = 3; Constant temperature using Langevin dynamics
- ig = -1; Seed for pseudo-random number generator
- ioutfm = 1; Write trajectory in netcdf format
- iwrap = 1; Wrap coords into primary box
- nstlim = 5000; Number of MD steps
- dt = 0.002; Time step (in ps)
- irest = 1; Restart previous simulation
- ntb = 1; Periodic boundary conditions at constant volume
- gamma_ln = 5.0; Collision frequency for Langevin dynamics (in 1/ps)

In this particular example, the **NVT equilibration** of the system is done in **500 steps** (note that the number of steps has been reduced in this tutorial, for the sake of time).

```
# Import module
from biobb_amber.sander.sander_mdrun import sander_mdrun

# Create prop dict and inputs/outputs
output_nvt_traj_path = 'sander.nvt.netcdf'
output_nvt_rst_path = 'sander.nvt.rst'
output_nvt_log_path = 'sander.nvt.log'

prop = {
    "simulation_type" : 'nvt',
    "mdin" : {
        'nstlim' : 500,      # Reducing the number of steps for the sake of time (1ps)
        'ntr' : 1,           # Turn on positional restraints
        'restraintmask' : '\"@CA,C,O,N\"',         # Restraining protein backbone atoms
        'restraint_wt' : 0.1                       # With a force constant of 0.1 Kcal/
↪mol*A2
    }
}

# Create and launch bb
sander_mdrun(input_top_path=output_top_cpin_path,
             input_crd_path=output_heat_rst_path,
             input_ref_path=output_heat_rst_path,
             output_traj_path=output_nvt_traj_path,
             output_rst_path=output_nvt_rst_path,
             output_log_path=output_nvt_log_path,
             properties=prop)
```

Step 2: Checking NVT Equilibration results

Checking **NVT Equilibration** results. Plotting **system temperature** by time during the NVT equilibration process.

```
# Import module
from biobb_amber.process.process_mdout import process_mdout

# Create prop dict and inputs/outputs
output_dat_nvt_path = 'sander.md.nvt.temp.dat'

prop = {
    "terms" : ['TEMP']
}

# Create and launch bb
process_mdout(input_log_path=output_nvt_log_path,
             output_dat_path=output_dat_nvt_path,
             properties=prop)
```

```
#Read data from file and filter energy values higher than 1000 Kj/mol^-1
with open(output_dat_nvt_path,'r') as energy_file:
    x,y = map(
```

(continues on next page)

(continued from previous page)

```
        list,
        zip(*[
            (float(line.split()[0]),float(line.split()[1]))
            for line in energy_file
            if not line.startswith(("#","@"))
            if float(line.split()[1]) < 1000
        ])
    )

plotly.offline.init_notebook_mode(connected=True)

fig = {
    "data": [go.Scatter(x=x, y=y)],
    "layout": go.Layout(title="NVT equilibration",
                        xaxis=dict(title = "Equilibration Step (ps)"),
                        yaxis=dict(title = "Temperature (K)")
                       )
}

plotly.offline.iplot(fig)
```

1.4.14 Equilibrate the system (NPT)

Equilibrate the **protein system** in **NPT ensemble** (constant Number of particles, Pressure and Temperature). Protein **backbone atoms** will be restrained using position restraining forces: movement is permitted, but only after overcoming a substantial energy penalty.

- *Step 1*: Equilibrate the **protein system** with **NPT** ensemble.
- *Step 2*: Checking **NPT Equilibration** results. Plotting **system pressure and density** by time during the **NVT equilibration** process.

Building Blocks used:

- sander_mdrun from **biobb_amber.sander.sander_mdrun**
- process_mdout from **biobb_amber.process.process_mdout**

Step 1: Equilibrating the system (NPT)

The **npt** type of the **simulation_type property** contains the main default parameters to run a **system equilibration in NPT ensemble**:

- imin = 0; Run MD (no minimization)
- ntx = 5; Read initial coords and vels from restart file
- cut = 10.0; Cutoff for non bonded interactions in Angstroms
- ntr = 0; No restrained atoms
- ntc = 2; SHAKE for constraining length of bonds involving Hydrogen atoms
- ntf = 2; Bond interactions involving H omitted
- ntt = 3; Constant temperature using Langevin dynamics
- ig = -1; Seed for pseudo-random number generator
- ioutfm = 1; Write trajectory in netcdf format
- iwrap = 1; Wrap coords into primary box
- nstlim = 5000; Number of MD steps
- dt = 0.002; Time step (in ps)
- irest = 1; Restart previous simulation
- gamma_ln = 5.0; Collision frequency for Langevin dynamics (in 1/ps)
- pres0 = 1.0; Reference pressure
- ntp = 1; Constant pressure dynamics: md with isotropic position scaling
- taup = 2.0; Pressure relaxation time (in ps)

In this particular example, the **NPT equilibration** of the system is done in **500 steps** (note that the number of steps has been reduced in this tutorial, for the sake of time).

```
# Import module
from biobb_amber.sander.sander_mdrun import sander_mdrun

# Create prop dict and inputs/outputs
output_npt_traj_path = 'sander.npt.netcdf'
output_npt_rst_path = 'sander.npt.rst'
output_npt_log_path = 'sander.npt.log'

prop = {
    "simulation_type" : 'npt',
    "mdin" : {
        'nstlim' : 500,      # Reducing the number of steps for the sake of time (1ps)
        'ntr' : 1,           # Turn on positional restraints
        'restraintmask' : '\"@CA,C,O,N\"',         # Restraining protein backbone atoms
        'restraint_wt' : 0.1                       # With a force constant of 0.1 Kcal/
↪mol*A2
    }
}

# Create and launch bb
```

(continues on next page)

(continued from previous page)

```
sander_mdrun(input_top_path=output_top_cpin_path,
             input_crd_path=output_nvt_rst_path,
             input_ref_path=output_nvt_rst_path,
             output_traj_path=output_npt_traj_path,
             output_rst_path=output_npt_rst_path,
             output_log_path=output_npt_log_path,
             properties=prop)
```

Step 2: Checking NPT Equilibration results

Checking **NPT Equilibration** results. Plotting **system pressure and density** by time during the **NPT equilibration** process.

```
# Import module
from biobb_amber.process.process_mdout import process_mdout

# Create prop dict and inputs/outputs
output_dat_npt_path = 'sander.md.npt.dat'

prop = {
    "terms" : ['PRES','DENSITY']
}

# Create and launch bb
process_mdout(input_log_path=output_npt_log_path,
             output_dat_path=output_dat_npt_path,
             properties=prop)
```

```
# Read pressure and density data from file
with open(output_dat_npt_path,'r') as pd_file:
    x,y,z = map(
        list,
        zip(*[
            (float(line.split()[0]),float(line.split()[1]),float(line.split()[2]))
            for line in pd_file
            if not line.startswith(("#","@"))
        ])
    )

plotly.offline.init_notebook_mode(connected=True)

trace1 = go.Scatter(
    x=x,y=y
)
trace2 = go.Scatter(
    x=x,y=z
)

fig = subplots.make_subplots(rows=1, cols=2, print_grid=False)
```

(continues on next page)

(continued from previous page)

```
fig.append_trace(trace1, 1, 1)
fig.append_trace(trace2, 1, 2)

fig['layout']['xaxis1'].update(title='Time (ps)')
fig['layout']['xaxis2'].update(title='Time (ps)')
fig['layout']['yaxis1'].update(title='Pressure (bar)')
fig['layout']['yaxis2'].update(title='Density (Kg*m^-3)')

fig['layout'].update(title='Pressure and Density during NPT Equilibration')
fig['layout'].update(showlegend=False)

plotly.offline.iplot(fig)
```

1.4.15 Constant pH Molecular Dynamics Simulation

Upon completion of the **two equilibration phases (NVT and NPT)**, the system is now well-equilibrated at the desired temperature and pressure. The **position restraints** can now be released. The last step of the **protein** MD setup is a short, **free MD simulation**, to ensure the robustness of the system.

- *Step 1*: Run short MD simulation of the **protein system**.
- *Step 2*: Checking results for the final step of the setup process, the **free MD run**. Plotting **Root Mean Square deviation (RMSd)** and **Radius of Gyration (Rgyr)** by time during the **free MD run** step.

Building Blocks used:

- sander_mdrun from **biobb_amber.sander.sander_mdrun**
- process_mdout from **biobb_amber.process.process_mdout**
- cpptraj_rms from **biobb_analysis.cpptraj.cpptraj_rms**
- cpptraj_rgyr from **biobb_analysis.cpptraj.cpptraj_rgyr**

Step 1: Creating portable binary run file to run a Constant pH MD simulation

The **free** type of the **simulation_type property** contains the main default parameters to run an **unrestrained MD simulation**:

- imin = 0; Run MD (no minimization)
- ntx = 5; Read initial coords and vels from restart file
- cut = 10.0; Cutoff for non bonded interactions in Angstroms
- ntr = 0; No restrained atoms

- ntc = 2; SHAKE for constraining length of bonds involving Hydrogen atoms
- ntf = 2; Bond interactions involving H omitted
- ntt = 3; Constant temperature using Langevin dynamics
- ig = -1; Seed for pseudo-random number generator
- ioutfm = 1; Write trajectory in netcdf format
- iwrap = 1; Wrap coords into primary box
- nstlim = 5000; Number of MD steps
- dt = 0.002; Time step (in ps)

In this particular example, a short, **5ps-length** simulation (2500 steps) is run, for the sake of time.

On top of these parameters, we will include the **constant pH** specific properties:

- icnstph = 2; Turn on constant pH for explicit solvent
- saltcon = 0.1; Use the salt concentration CpHMD was parameterized for
- ntcnstph = 100; Protonation state change attempt every 100 steps
- ntrelax = 100; Number of relaxation steps after a successful protonation state change
- solvph = 7.0; Solvent pH

```
# Import module
from biobb_amber.sander.sander_mdrun import sander_mdrun

# Create prop dict and inputs/outputs
output_pH_traj_path = 'sander.pH.netcdf'
output_pH_rst_path = 'sander.pH.rst'
output_pH_cpout_path = 'sander.pH.cpout'
output_pH_cprst_path = 'sander.pH.cprst'
output_pH_log_path = 'sander.pH.log'
output_pH_mdinfo_path = 'sander.pH.mdinfo'

prop = {
    "simulation_type" : 'free',
    "mdin" : {
        'nstlim' : 2500,      # Reducing the number of steps for the sake of time (5ps)
        'ntwx' : 500,         # Print coords to trajectory every 500 steps (1 ps)
        'icnstph' : 2,        # Turn on constant pH for explicit solvent
        'saltcon' : 0.1,      # Use the salt concentration CpHMD was parameterized for
        'ntcnstph' : 100,     # Protonation state change attempt every 100 steps
        'ntrelax' : 100,      # Number of relaxation steps after a successful protonation␣
↪state change
        'solvph' : 7.0,       # Solvent pH
#        'solvph' : 3.0,       # Acid pH
#        'solvph' : 10.0,      # Basic (alkaline) pH
    }
}

# Create and launch bb
sander_mdrun(input_top_path=output_top_cpin_path,
             input_crd_path=output_npt_rst_path,
```

(continues on next page)

(continued from previous page)

```
            input_cpin_path=output_cpin_path,
            output_traj_path=output_pH_traj_path,
            output_rst_path=output_pH_rst_path,
            output_cpout_path=output_pH_cpout_path,
            output_cprst_path=output_pH_cprst_path,
            output_log_path=output_pH_log_path,
            output_mdinfo_path=output_pH_mdinfo_path,
            properties=prop)
```

Step 2: Checking constant pH MD simulation results

Protonation states that are sampled throughout the course of the **constant pH** simulations are written to a **cpout-formatted file**. The program **cphstats** can be used to parse this **cpout file** and extract the **predicted pKa values** along different parameters:

- The difference between the predicted pKa and the system pH (**Offset**)
- The predicted pKa (**Pred**)
- The fraction of time the residue spends protonated (**Frac Prot**)
- The number of accpeted protonations state transitions (**Transitions**)
- The sum of the fractional protonations (**Average total molecular protonation**)

An additional **population** file is generated, containing the populations of every state for every **titratable residue**, the fraction of snapshots that the system spent in each state for each residue.

```
# Import module
from biobb_amber.cphstats.cphstats_run import cphstats_run

# Create prop dict and inputs/outputs
output_pH_dat_path = 'cphstats.pH.dat'
output_pH_pop_path = 'cphstats.pH.pop.dat'

prop = {
    'verbose' : True,
    'running_avg_window' : 1
}

# Create and launch bb
cphstats_run(input_cpin_path=output_cpin_path,
            input_cpout_path=output_pH_cpout_path,
            output_dat_path=output_pH_dat_path,
            output_population_path=output_pH_pop_path,
            properties=prop)
```

Last Remarks

When checking the information from the **predicted pKa values** cphstats.pH.dat and the **state population** cphstats.pH.pop.dat coming from the **constant pH** simulation at physiological pH (~7), you will find that no different states other than the major species appear during the simulation. Try to re-run the **constant pH** simulation again, modifying the pH parameter, using acid (<7) or basic (>7) pH and analyse again the results of the checking (cphstats) step.

An additional recommended and useful study is to repeat the **constant pH** simulation done in the previous step with different pH values (typically from 0 to 14), and then use the output **deprotonated fractions** for each residue as a function of the pH to plot **titration curves**. See the AMBER tutorial nº18 or AMBER tutorial nº33 for more information.

1.4.16 Output files

Important **Output files** generated:

- cphstats.pH.dat: **Predicted pKa values** extracted from the constant pH MD simulation.
- cphstats.pH.pop.dat: **Populations** of every state for every **titratable residue**, fraction of snapshots that the system spent in each state for each residue.

1.4.17 Questions & Comments

Questions, issues, suggestions and comments are really welcome!

- GitHub issues:
  - https://github.com/bioexcel/biobb
- BioExcel forum:
  - https://ask.bioexcel.eu/c/BioExcel-Building-Blocks-library

1.5 ABC MD Setup pipeline using BioExcel Building Blocks (biobb)

This **BioExcel Building Blocks library (BioBB) workflow** provides a pipeline to setup DNA structures for the **Ascona B-DNA Consortium** (ABC) members. It follows the work started with the NAFlex tool to offer a single, reproducible pipeline for structure preparation, ensuring **reproducibility** and **coherence** between all the members of the consortium. The **NAFlex pipeline** was used for the preparation of all the simulations done in the study: ***The static and dynamic structural heterogeneities of B-DNA: extending Calladine–Dickerson rules***. The workflow included in this **Jupyter Notebook** is **extending** and **updating** the **NAFlex pipeline**, following the best practices exposed by the *Daniel R. Roe and Bernard R. Brooks* work, and is being used for the **new ABC study**.

The **setup process** is performed using the **biobb_amber** module from the **BioBB library**, which is wrapping the **AMBER MD package**. The forcefield used is the nucleic acids specific **parmbsc1 forcefield**, with **Joung & Cheatham** monovalent ion parameters for **Ewald** and **TIP4P/EW** water and **SPC/E Water model**.

The main **steps of the pipeline** are:

- **Model** the **B-DNA** structure

- Generate structure **topology**
- **Solvate** structure with a truncated octahedron box, with **SCP/E water model**
- **Neutralize** the system with Potassium ions
- Add an **ionic concentration** of 150mM of Cl- / K+ ions
- **Randomize ions** around the structure using cpptraj
- Generate **H-mass repartitioned topology** to run the production simulations with a 4fs timestep
- **Equilibrate** the system in solvent with a 10-steps protocol (Daniel R. Roe and Bernard R. Brooks)
- **Production Run** (4fs timestep)

1.5.1 Settings

Biobb module used

- biobb_amber: Tools to setup and run Molecular Dynamics simulations using the AMBER MD package.

Auxiliar libraries used

- nb_conda_kernels: Enables a Jupyter Notebook or JupyterLab application in one conda environment to access kernels for Python, R, and other languages found in other environments.
- nglview: Jupyter/IPython widget to interactively view molecular structures and trajectories in notebooks.
- ipywidgets: Interactive HTML widgets for Jupyter notebooks and the IPython kernel.
- plotly: Python interactive graphing library integrated in Jupyter notebooks.

Conda Installation and Launch

```
git clone https://github.com/bioexcel/biobb_wf_amber_md_setup.git
cd biobb_wf_md_setup_amber
conda env create -f conda_env/environment.yml
conda activate biobb_MDsetup_tutorials_amber
./conda_env/post-link.sh
jupyter-notebook biobb_wf_amber_md_setup/notebooks/abcsetup/biobb_amber_ABC_setup.ipynb
```

1.5.2 Pipeline steps

1. *Initial Parameters*
2. *Model DNA 3D Structure*
3. *Generate Topology*
4. *Add Water Box*
5. *Adding additional ionic concentration*

6. *Randomize Ions*
7. *Generate Topology with Hydrogen Mass Partitioning (4fs)*
8. *System Equilibration*
9. *Free MD Simulation*
10. *Output files*

1.5.3 Auxiliar libraries

```
import nglview
import ipywidgets
import plotly
import plotly.graph_objs as go
```

1.5.4 Initial parameters

Input parameters needed:

- **DNA sequence**: Nucleotide sequence to be modelled and prepared for a MD simulation (e.g. GCGCGGCTGATAAACGAAAGC)
- **Forcefield**: Forcefield to be used in the setup (e.g. protein.ff14SB).
- **Water model**: Water model to be used in the setup (e.g. SPC/E).
- **Ion model**: Ion model to be used in the setup (e.g. Dang).
- **Thermostat**: Thermostat to be used in the setup (e.g. Langevin).
- **Timestep**: Simulation timestep (e.g 2fs).

```
seq = "CGCGAATTCGCG" # Drew-Dickerson dodecamer

forcefield = ["DNA.bsc1"] # ParmBSC1 (ff99 + bsc0 + bsc1) for DNA. Ivani et al. Nature␣
↪Methods 13: 55, 2016
water_model = "OPCBOX" # SPC/E + Joung-Chetham monovalent ions + Li/Merz highly charged␣
↪ions (+2 to +4, 12-6 normal usage set)
ions_model = "ionsjc_tip4pew" # Monovalent ion parameters for Ewald and TIP4P/EW water␣
↪from Joung & Cheatham JPCB (2008)
```

1.5.5 Model DNA 3D structure

Model **DNA 3D structure** from a **nucleotide sequence** using the **nab tool** from the **AMBER MD package**.

Building Blocks used:

- nab_build_dna_structure from **biobb_amber.nab.nab_build_dna_structure**

```
# Import module
from biobb_amber.nab.nab_build_dna_structure import nab_build_dna_structure

# Create properties dict and inputs/outputs
dna_pdb = seq+'.pdb'
prop = {
    'sequence': seq,
    'helix_type': 'abdna', # Right Handed B-DNA, Arnott
    'remove_tmp': True,
    'compiler': 'gcc', # change according to your operating system
    'linker': 'x86_64-conda_cos6-linux-gnu-gfortran' # gfortran linux version
    # 'linker': 'gfortran' # gfortran osx version
}

#Create and launch bb
nab_build_dna_structure(output_pdb_path=dna_pdb,
    properties=prop)
```

Visualizing 3D structure

```
# Show protein
view = nglview.show_structure_file(dna_pdb)
view.add_representation(repr_type='ball+stick', selection='all')
view._remote_call('setSize', target='Widget', args=['','600px'])
view
```

1.5.6 Generate Topology

Build the **DNA topology** from the **modelled structure** using the **leap tool** from the **AMBER MD package**. Using the **forcefield** fixed in the first cell.

Building Blocks used:

- leap_gen_top from **biobb_amber.leap.leap_gen_top**

```
# Import module
from biobb_amber.leap.leap_gen_top import leap_gen_top

# Create prop dict and inputs/outputs
prop = {
    "forcefield" : forcefield
}
dna_leap_pdb_path = 'structure.leap.pdb'
dna_leap_top_path = 'structure.leap.top'
dna_leap_crd_path = 'structure.leap.crd'

# Create and launch bb
leap_gen_top(input_pdb_path=dna_pdb,
           output_pdb_path=dna_leap_pdb_path,
           output_top_path=dna_leap_top_path,
           output_crd_path=dna_leap_crd_path,
           properties=prop)
```

```
# Show protein
view = nglview.show_structure_file(dna_leap_pdb_path)
view.add_representation(repr_type='ball+stick', selection='all')
view._remote_call('setSize', target='Widget', args=['','600px'])
view
```

1.5.7 Adding Water Box

Creating a **water box** surrounding the **DNA structure** using the **leap tool** from the **AMBER MD package**. Using the **water model** fixed in the first cell.

Building Blocks used:

- amber_to_pdb from **biobb_amber.ambpdb.amber_to_pdb**
- leap_solvate from **biobb_amber.leap.leap_solvate**

Add water box

Define the **unit cell** for the **DNA structure MD system** and fill it with **water molecules**. A **truncated octahedron** is used to define the unit cell, with a distance from the protein to the box edge of 15Å. The **water model** used is the one defined in the first cell.

```
# Import module
from biobb_amber.leap.leap_solvate import leap_solvate

# Create prop dict and inputs/outputs
prop = {
```

(continues on next page)

(continued from previous page)

```
    'forcefield': forcefield,
    'water_type': water_model,
    'ions_type' : ions_model,
    'box_type': 'truncated_octahedron',
    'distance_to_molecule' : 15.0,
    'neutralise' : True,
    'iso' : True,
    'closeness' : 0.97,
    'positive_ions_type' : "K+"
}

output_solv_pdb_path = 'structure.solv.pdb'
output_solv_top_path = 'structure.solv.parmtop'
output_solv_crd_path = 'structure.solv.crd'

# Create and launch bb
leap_solvate( input_pdb_path=dna_leap_pdb_path,
            output_pdb_path=output_solv_pdb_path,
            output_top_path=output_solv_top_path,
            output_crd_path=output_solv_crd_path,
            properties=prop)
```

```
# Show protein
view = nglview.show_structure_file(output_solv_pdb_path)
view.clear_representations()
view.add_representation(repr_type='ball+stick', selection='nucleic')
view.add_representation(repr_type='line', selection='water')
view._remote_call('setSize', target='Widget', args=['','600px'])
view
```

1.5.8 Adding additional ionic concentration

Neutralizing the system and adding an additional **ionic concentration** using the **leap tool** from the **AMBER MD package**. Using **Potassium (K+)** and **Chloride (Cl-)** counterions and an **additional ionic concentration** of 150mM.

Building Blocks used:

- leap_add_ions from **biobb_amber.leap.leap_add_ions**

```
# Import module
from biobb_amber.leap.leap_add_ions import leap_add_ions

# Create prop dict and inputs/outputs
prop = {
    'forcefield': forcefield,
```

(continues on next page)

(continued from previous page)

```
    'water_type': water_model,
    'ions_type' : ions_model,
    'box_type': 'truncated_octahedron',
    'ionic_concentration' : 100,  # 100 Mol/L
    'positive_ions_type' : "K+"
}

output_ions_pdb_path = 'structure.ions.pdb'
output_ions_top_path = 'structure.ions.parmtop'
output_ions_crd_path = 'structure.ions.crd'

# Create and launch bb
leap_add_ions(input_pdb_path=output_solv_pdb_path,
            output_pdb_path=output_ions_pdb_path,
            output_top_path=output_ions_top_path,
            output_crd_path=output_ions_crd_path,
            properties=prop)
```

```
# Show protein
view = nglview.show_structure_file(output_ions_pdb_path)
view.clear_representations()
view.add_representation(repr_type='ball+stick', selection='nucleic')
view.add_representation(repr_type='spacefill', selection='Na+')
view.add_representation(repr_type='spacefill', selection='K+')
view.add_representation(repr_type='spacefill', selection='Cl-')
view._remote_call('setSize', target='Widget', args=['','600px'])
view
```

1.5.9 Randomize ions

Randomly swap the positions of **solvent** and **ions** using the **cpptraj tool** from the **AMBER MD package**.

Building Blocks used:

- cpptraj_randomize_ions from **biobb_amber.cpptraj.cpptraj_randomize_ions**

```
# Import module
from biobb_amber.cpptraj.cpptraj_randomize_ions import cpptraj_randomize_ions

# Create prop dict and inputs/outputs
prop = {
    "distance" : 6.0,
    "overlap" : 4.0
}
output_cpptraj_crd_path = 'structure.randIons.crd'
```

(continues on next page)

(continued from previous page)

```
output_cpptraj_pdb_path = 'structure.randIons.pdb'

# Create and launch bb
cpptraj_randomize_ions(
            input_top_path=output_ions_top_path,
            input_crd_path=output_ions_crd_path,
            output_pdb_path=output_cpptraj_pdb_path,
            output_crd_path=output_cpptraj_crd_path,
            properties=prop)
```

```
# Show protein
view = nglview.show_structure_file(output_cpptraj_pdb_path)
view.clear_representations()
view.add_representation(repr_type='ball+stick', selection='nucleic')
view.add_representation(repr_type='spacefill', selection='K+')
view.add_representation(repr_type='spacefill', selection='Cl-')
view._remote_call('setSize', target='Widget', args=['','600px'])
view
```

1.5.10 Generate Topology with Hydrogen Mass Partitioning (4fs)

Modifying the **DNA topology** from the **modelled structure**, tripling the mass of all hydrogens on the system and scaling down the mass of all other atoms using the **parmed tool** from the **AMBER MD package**.

Building Blocks used:

- parmed_hmassrepartition from **biobb_amber.parmed.parmed_hmassrepartition**

```
# Import module
from biobb_amber.parmed.parmed_hmassrepartition import parmed_hmassrepartition

# Create prop dict and inputs/outputs
dna_leap_top_4fs_path = 'structure.leap.4fs.top'

# Create and launch bb
parmed_hmassrepartition(
    input_top_path=output_ions_top_path,
    output_top_path=dna_leap_top_4fs_path
)
```

1.5.11 Standard Equilibration for explicit solvent

With the **DNA + solvent + counterions** system ready, the next step in the **MD setup** is the **system equilibration**. In this step, atoms of the macromolecules and of the surrounding solvent undergo a relaxation that usually lasts for tens or hundreds of picoseconds before the system reaches a stationary state (see Amber tutorials).

Many different **equilibration protocols** exist. The protocol included in this workflow was prepared, tested and compared with different conditions and DNA sequences by the **ABC consortium**, and finally chosen as the **standard protocol** for the **2021 ABCix run**. It is based on the *Daniel R. Roe and Bernard R. Brooks* work (*A protocol for preparing explicitly solvated systems for stable molecular dynamics simulations*) and comprises **10 different stages**:

1. *Equilibration Step 1* – System Energetic Minimization, 5 Kcal/mol heavy atoms restraints (1000 cycles)
2. *Equilibration Step 2* – NVT Equilibration, 5 Kcal/mol heavy atoms restraints, timestep 1fs (15ps)
3. *Equilibration Step 3* – System Energetic Minimization, 2 Kcal/mol heavy atoms restraints (1000 cycles)
4. *Equilibration Step 4* – System Energetic Minimization, 0.1 Kcal/mol heavy atoms restraints (1000 cycles)
5. *Equilibration Step 5* – System Energetic Minimization (1000 cycles)
6. *Equilibration Step 6* – NPT Equilibration, 1 Kcal/mol heavy atoms restraints, timestep 1fs (5ps)
7. *Equilibration Step 7* – NPT Equilibration, 0.5 Kcal/mol heavy atoms restraints, timestep 1fs (5ps)
8. *Equilibration Step 8* – NPT Equilibration, 0.5 Kcal/mol backbone atoms restraints, timestep 1fs (10ps)
9. *Equilibration Step 9* – NPT Equilibration, timestep 2fs (10ps)
10. *Equilibration Step 10* – NPT Equilibration, timestep 2fs, long simulation (1ns)

Equilibration Step 1: System energetic minimization

Energetically minimize the **DNA structure** (in solvent) using the **sander tool** from the **AMBER MD package**. Relaxing **solvent** molecules around the **DNA structure**.

AMBER MD configuration file used (step1.in) includes the following **simulation parameters**:

- imin = 1; Run minimization
- ntmin = 2; Steepest Descent minimization method
- maxcyc = 1000; Number of minimization steps
- ncyc = 10; Switch from steepest descent to conjugate gradient after 10 cycles (if ntmin = 1)
- ntwx = 500; Coordinates will be written to the output trajectory file every 500 steps
- ioutfm = 1; Write trajectory in netcdf format
- ntxo = 2; Write restart in netcdf format
- ntpr = 50; Write energy information to files 'mdout' and 'mdinfo' every 50 steps
- ntwr = 500; Write information to restart file every 500 steps
- ntc = 1; Turn off SHAKE for constraining length of bonds involving Hydrogen atoms
- ntf = 1; Force evaluation: complete interactions are calculated
- ntb = 1; Constant Volume Periodic Boundary Conditions (PBC)
- cut = 8.0; Cutoff for non bonded interactions in Angstroms
- ntr = 1; Turn on positional restraints

- restraintmask = :1-40&!@H=; Restraints on DNA atoms only
- restraint_wt = 5.0; Restraint force constant

Minimization step applying **restraints** on the **DNA heavy atoms** with a **force constant** of **5 Kcal/mol.$Å^{2}$**

Building Blocks used:

- sander_mdrun from **biobb_amber.sander.sander_mdrun**
- process_minout from **biobb_amber.process.process_minout**

```
# Import module
from biobb_amber.sander.sander_mdrun import sander_mdrun

# Create prop dict and inputs/outputs
prop = {
    "mdin" : {
        'maxcyc' : 500,  # Overwrite number of minimization steps if needed
        'restraintmask' : '\":DA,DC,DG,DT,D=3,D=5&!@H=\"' # Overwrite DNA heavy atoms␣
↪mask to make it more generic
    },
#    "sander_path" : "sander.MPI",  # Change sander binary to parallel (MPI) execution␣
↪(not included in AmberTools)
#    "mpi_bin" : "mpirun",          # MPI runner
#    "mpi_np" : 16                  # Number of cores to use in the MPI parallel␣
↪calculation
}
output_eq1_traj_path = 'sander.eq1.nc'
output_eq1_rst_path = 'sander.eq1.ncrst'
output_eq1_log_path = 'sander.eq1.log'
output_eq1_mdinfo_path = 'sander.eq1.mdinfo'

# Create and launch bb
sander_mdrun(
            input_top_path=dna_leap_top_4fs_path,
            input_mdin_path="ABCix_config_files/step1.in",
            input_crd_path=output_cpptraj_crd_path,
            input_ref_path=output_cpptraj_crd_path,
            output_traj_path=output_eq1_traj_path,
            output_rst_path=output_eq1_rst_path,
            output_mdinfo_path=output_eq1_mdinfo_path,
            output_log_path=output_eq1_log_path,
            properties=prop)
```

Checking Equilibration Step 1 results

Checking **Equilibration Step 1 - System Energetic Minimization** results. Plotting **potential energy** along time during the **minimization process**.

```
# Import module
from biobb_amber.process.process_minout import process_minout

# Create prop dict and inputs/outputs
prop = {
    "terms" : ['ENERGY'],
    "remove_tmp": True
}
output_dat_eq1_path = 'sander.eq1.energy.dat'

# Create and launch bb
process_minout(input_log_path=output_eq1_log_path,
            output_dat_path=output_dat_eq1_path,
            properties=prop)
```

```
#Read data from file and filter energy values higher than 1000 Kj/mol^-1
with open(output_dat_eq1_path,'r') as energy_file:
    x,y = map(
        list,
        zip(*[
            (float(line.split()[0]),float(line.split()[1]))
            for line in energy_file
            if not line.startswith(("#","@"))
            if float(line.split()[1]) < 1000
        ])
    )

plotly.offline.init_notebook_mode(connected=True)

fig = {
    "data": [go.Scatter(x=x, y=y)],
    "layout": go.Layout(title="Equilibration Step 1",
                        xaxis=dict(title = "Energy Minimization Step"),
                        yaxis=dict(title = "Potential Energy kcal/mol")
                       )
}

plotly.offline.iplot(fig)
```

Equilibration Step 2: NVT equilibration

Equilibrate the **system** in **NVT** ensemble (constant number of particles -N-, Volume -V-, and Temperature -T-), using the **sander tool** from the **AMBER MD package**. Taking the system to the desired **temperature**.

AMBER MD configuration file used (step2.in) includes the following **simulation parameters**:

- imin = 0; Run MD (no minimization)
- nstlim = 15000; Number of MD steps
- dt = 0.001; Time step (in ps)
- ntx = 1; Only read initial coordinates from input files
- irest = 0; Do not restart a simulation, start a new one
- ig = -1; Seed for the pseudo-random number generator
- ntwx = 500; Coordinates will be written to the output trajectory file every 500 steps
- ntwv = -1; Velocities will be written to output trajectory file, making it a combined coordinate/velocity trajectory file, at the interval defined by ntwx
- ioutfm = 1; Write trajectory in netcdf format
- ntxo = 2; Write restart in netcdf format
- ntpr = 50; Write energy information to files 'mdout' and 'mdinfo' every 50 steps
- ntwr = 500; Write information to restart file every 500 steps
- iwrap = 0; Not wrapping coordinates into primary box
- nscm = 0; Not removing translational and rotational center-of-mass motions
- ntc = 2; Turn on SHAKE for constraining length of bonds involving Hydrogen atoms
- ntf = 2; Force evaluation: Bond interactions involving H omitted (SHAKE)
- ntb = 1; Constant Volume Periodic Boundary Conditions (PBC)
- cut = 8.0; Cutoff for non bonded interactions in Angstroms
- ntt = 3; Constant temperature using Langevin dynamics
- gamma_ln = 5; Collision frequency for Langevin dynamics (in 1/ps)
- temp0 = 310.0; Final temperature (310 K)
- tempi = 310.0; Initial temperature (310 K)
- ntr = 1; Turn on positional restraints
- restraintmask = :1-40&!@H=; Restraints on DNA atoms only
- restraint_wt = 5.0; Restraint force constant

NVT equilibration step applying **restraints** on the **DNA heavy atoms** with a **force constant** of **5 Kcal/mol.$Å^{2}$**

Building Blocks used:

- sander_mdrun from **biobb_amber.sander.sander_mdrun**
- process_mdout from **biobb_amber.process.process_mdout**

```
# Import module
from biobb_amber.sander.sander_mdrun import sander_mdrun

# Create prop dict and inputs/outputs
prop = {
    "mdin" : {
        'nstlim' : 500,  # Overwrite number of MD steps if needed
        'restraintmask' : '\":DA,DC,DG,DT,D=3,D=5&!@H=\"' # Overwrite DNA heavy atoms␣
↪mask to make it more generic
    },
#    "sander_path" : "sander.MPI",  # Change sander binary to parallel (MPI) execution␣
↪(not included in AmberTools)
#    "mpi_bin" : "mpirun",          # MPI runner
#    "mpi_np" : 16                  # Number of cores to use in the MPI parallel␣
↪calculation
}
output_eq2_traj_path = 'sander.eq2.nc'
output_eq2_rst_path = 'sander.eq2.ncrst'
output_eq2_log_path = 'sander.eq2.log'
output_eq2_mdinfo_path = 'sander.eq2.mdinfo'

# Create and launch bb
sander_mdrun(input_top_path=dna_leap_top_4fs_path,
             input_mdin_path="ABCix_config_files/step2.in",
             input_crd_path=output_eq1_rst_path,
             input_ref_path=output_eq1_rst_path,
             output_traj_path=output_eq2_traj_path,
             output_rst_path=output_eq2_rst_path,
             output_log_path=output_eq2_log_path,
             output_mdinfo_path=output_eq2_mdinfo_path,
             properties=prop)
```

Checking Equilibration Step 2 results

Checking **Equilibration Step 2 - NVT Equilibration** results. Plotting **temperature** by time during the **equilibration process**.

```
# Import module
from biobb_amber.process.process_mdout import process_mdout

# Create prop dict and inputs/outputs
prop = {
    "terms" : ['TEMP']
}
output_dat_eq2_path = 'sander.eq2.energy.dat'

# Create and launch bb
process_mdout(input_log_path=output_eq2_log_path,
              output_dat_path=output_dat_eq2_path,
              properties=prop)
```

```
#Read data from file and filter energy values higher than 1000 Kj/mol^-1
with open(output_dat_eq2_path,'r') as energy_file:
    x,y = map(
        list,
        zip(*[
            (float(line.split()[0]),float(line.split()[1]))
            for line in energy_file
            if not line.startswith(("#","@"))
            if float(line.split()[1]) < 1000
        ])
    )

plotly.offline.init_notebook_mode(connected=True)

fig = {
    "data": [go.Scatter(x=x, y=y)],
    "layout": go.Layout(title="Equilibration Step 2 - NVT equilibration",
                        xaxis=dict(title = "Energy equilibration time (ps)"),
                        yaxis=dict(title = "Temperature (K)")
                       )
}

plotly.offline.iplot(fig)
```

Equilibration Step 3: System energetic minimization

Energetically minimize the **DNA structure** (in solvent) using the **sander tool** from the **AMBER MD package**. Relaxing **system** with **soft restraints** on the **DNA structure**.

AMBER MD configuration file used (step3.in) includes the following **simulation parameters**:

- imin = 1; Run minimization
- ntmin = 2; Steepest Descent minimization method
- maxcyc = 1000; Number of minimization steps
- ncyc = 10; Switch from steepest descent to conjugate gradient after 10 cycles (if ntmin = 1)
- ntwx = 500; Coordinates will be written to the output trajectory file every 500 steps
- ioutfm = 1; Write trajectory in netcdf format
- ntxo = 2; Write restart in netcdf format
- ntpr = 50; Write energy information to files 'mdout' and 'mdinfo' every 50 steps
- ntwr = 500; Write information to restart file every 500 steps
- ntc = 1; Turn off SHAKE for constraining length of bonds involving Hydrogen atoms
- ntf = 1; Force evaluation: complete interactions are calculated
- ntb = 1; Constant Volume Periodic Boundary Conditions (PBC)
- cut = 8.0; Cutoff for non bonded interactions in Angstroms

- ntr = 1; Turn on positional restraints
- restraintmask = :1-40&!@H=; Restraints on DNA atoms only
- restraint_wt = 2.0; Restraint force constant

Further **minimization step** lowering the **restraints** on the **DNA heavy atoms** to **2 Kcal/mol.$Å^{2}$** force constant.

Building Blocks used:

- sander_mdrun from **biobb_amber.sander.sander_mdrun**
- process_minout from **biobb_amber.process.process_minout**

```
# Import module
from biobb_amber.sander.sander_mdrun import sander_mdrun

# Create prop dict and inputs/outputs
prop = {
    "mdin" : {
        'maxcyc' : 500,  # Overwrite number of minimization steps if needed
        'restraintmask' : '\":DA,DC,DG,DT,D=3,D=5&!@H=\"' # Overwrite DNA heavy atoms␣
↪mask to make it more generic
    },
#    "sander_path" : "sander.MPI",  # Change sander binary to parallel (MPI) execution␣
↪(not included in AmberTools)
#    "mpi_bin" : "mpirun",          # MPI runner
#    "mpi_np" : 16                  # Number of cores to use in the MPI parallel␣
↪calculation
}
output_eq3_traj_path = 'sander.eq3.nc'
output_eq3_rst_path = 'sander.eq3.ncrst'
output_eq3_log_path = 'sander.eq3.log'
output_eq3_mdinfo_path = 'sander.eq3.mdinfo'

# Create and launch bb
sander_mdrun(input_top_path=dna_leap_top_4fs_path,
             input_mdin_path="ABCix_config_files/step3.in",
             input_crd_path=output_eq2_rst_path,
             input_ref_path=output_eq2_rst_path,
             output_traj_path=output_eq3_traj_path,
             output_rst_path=output_eq3_rst_path,
             output_log_path=output_eq3_log_path,
             output_mdinfo_path=output_eq3_mdinfo_path,
             properties=prop)
```

Checking Equilibration Step 3 results

Checking **Equilibration Step 3 - System Energetic Minimization** results. Plotting **potential energy** along time during the **minimization process**.

```
# Import module
from biobb_amber.process.process_minout import process_minout

# Create prop dict and inputs/outputs
prop = {
    "terms" : ['ENERGY']
}
output_dat_eq3_path = 'sander.eq3.energy.dat'

# Create and launch bb
process_minout(input_log_path=output_eq3_log_path,
            output_dat_path=output_dat_eq3_path,
            properties=prop)
```

```
#Read data from file and filter energy values higher than 1000 Kj/mol^-1
with open(output_dat_eq3_path,'r') as energy_file:
    x,y = map(
        list,
        zip(*[
            (float(line.split()[0]),float(line.split()[1]))
            for line in energy_file
            if not line.startswith(("#","@"))
            if float(line.split()[1]) < 1000
        ])
    )

plotly.offline.init_notebook_mode(connected=True)

fig = {
    "data": [go.Scatter(x=x, y=y)],
    "layout": go.Layout(title="Equilibration Step 3",
                        xaxis=dict(title = "Energy Minimization Step"),
                        yaxis=dict(title = "Potential Energy kcal/mol")
                       )
}

plotly.offline.iplot(fig)
```

Equilibration Step 4: System energetic minimization

Energetically minimize the **DNA structure** (in solvent) using the **sander tool** from the **AMBER MD package**. Relaxing **system** with **minimum restraints** on the **DNA structure**.

AMBER MD configuration file used (step4.in) includes the following **simulation parameters**:

- imin = 1; Run minimization
- ntmin = 2; Steepest Descent minimization method
- maxcyc = 1000; Number of minimization steps
- ncyc = 10; Switch from steepest descent to conjugate gradient after 10 cycles (if ntmin = 1)
- ntwx = 500; Coordinates will be written to the output trajectory file every 500 steps
- ioutfm = 1; Write trajectory in netcdf format
- ntxo = 2; Write restart in netcdf format
- ntpr = 50; Write energy information to files 'mdout' and 'mdinfo' every 50 steps
- ntwr = 500; Write information to restart file every 500 steps
- ntc = 1; Turn off SHAKE for constraining length of bonds involving Hydrogen atoms
- ntf = 1; Force evaluation: complete interactions are calculated
- ntb = 1; Constant Volume Periodic Boundary Conditions (PBC)
- cut = 8.0; Cutoff for non bonded interactions in Angstroms
- ntr = 1; Turn on positional restraints
- restraintmask = :1-40&!@H=; Restraints on DNA atoms only
- restraint_wt = 0.1; Restraint force constant

Further **minimization step** lowering the **restraints** on the **DNA heavy atoms** to **0.1 Kcal/mol.$Å^{2}$** force constant.

Building Blocks used:

- sander_mdrun from **biobb_amber.sander.sander_mdrun**
- process_minout from **biobb_amber.process.process_minout**

```
# Import module
from biobb_amber.sander.sander_mdrun import sander_mdrun

# Create prop dict and inputs/outputs
prop = {
    "mdin" : {
        'maxcyc' : 500,  # Overwrite number of minimization steps if needed
        'restraintmask' : '\":DA,DC,DG,DT,D=3,D=5&!@H=\"' # Overwrite DNA heavy atoms␣
↪mask to make it more generic
    },
#    "sander_path" : "sander.MPI",  # Change sander binary to parallel (MPI) execution␣
↪(not included in AmberTools)
#    "mpi_bin" : "mpirun",          # MPI runner
```

(continues on next page)

(continued from previous page)

```
#    "mpi_np" : 16                  # Number of cores to use in the MPI parallel␣
↪calculation
}
output_eq4_traj_path = 'sander.eq4.nc'
output_eq4_rst_path = 'sander.eq4.ncrst'
output_eq4_log_path = 'sander.eq4.log'
output_eq4_mdinfo_path = 'sander.eq4.mdinfo'

# Create and launch bb
sander_mdrun(input_top_path=dna_leap_top_4fs_path,
             input_mdin_path="ABCix_config_files/step4.in",
             input_crd_path=output_eq3_rst_path,
             input_ref_path=output_eq3_rst_path,
             output_traj_path=output_eq4_traj_path,
             output_rst_path=output_eq4_rst_path,
             output_log_path=output_eq4_log_path,
             output_mdinfo_path=output_eq4_mdinfo_path,
             properties=prop)
```

Checking Equilibration Step 4 results

Checking **Equilibration Step 4 - System Energetic Minimization** results. Plotting **potential energy** along time during the **minimization process**.

```
# Import module
from biobb_amber.process.process_minout import process_minout

# Create prop dict and inputs/outputs
prop = {
    "terms" : ['ENERGY']
}
output_dat_eq4_path = 'sander.eq4.energy.dat'

# Create and launch bb
process_minout(input_log_path=output_eq4_log_path,
             output_dat_path=output_dat_eq4_path,
             properties=prop)
```

```
#Read data from file and filter energy values higher than 1000 Kj/mol^-1
with open(output_dat_eq4_path,'r') as energy_file:
    x,y = map(
        list,
        zip(*[
            (float(line.split()[0]),float(line.split()[1]))
            for line in energy_file
            if not line.startswith(("#","@"))
            if float(line.split()[1]) < 1000
        ])
    )

plotly.offline.init_notebook_mode(connected=True)
```

(continues on next page)

(continued from previous page)

```
fig = {
    "data": [go.Scatter(x=x, y=y)],
    "layout": go.Layout(title="Equilibration Step 4",
                        xaxis=dict(title = "Energy Minimization Step"),
                        yaxis=dict(title = "Potential Energy kcal/mol")
                       )
}

plotly.offline.iplot(fig)
```

Equilibration Step 5: System energetic minimization

Energetically minimize the **DNA structure** (in solvent) using the **sander tool** from the **AMBER MD package**. Relaxing **system without restraints** on the **DNA structure**.

AMBER MD configuration file used (step5.in) includes the following **simulation parameters**:

- imin = 1; Run minimization
- ntmin = 2; Steepest Descent minimization method
- maxcyc = 1000; Number of minimization steps
- ncyc = 10; Switch from steepest descent to conjugate gradient after 10 cycles (if ntmin = 1)
- ntwx = 500; Coordinates will be written to the output trajectory file every 500 steps
- ioutfm = 1; Write trajectory in netcdf format
- ntxo = 2; Write restart in netcdf format
- ntpr = 50; Write energy information to files 'mdout' and 'mdinfo' every 50 steps
- ntwr = 500; Write information to restart file every 500 steps
- ntc = 1; Turn off SHAKE for constraining length of bonds involving Hydrogen atoms
- ntf = 1; Force evaluation: complete interactions are calculated
- ntb = 1; Constant Volume Periodic Boundary Conditions (PBC)
- cut = 8.0; Cutoff for non bonded interactions in Angstroms
- ntr = 0; Turn off positional restraints

Further **minimization step**, with all **position restraints** on the **DNA heavy atoms** released.

Building Blocks used:

- sander_mdrun from **biobb_amber.sander.sander_mdrun**
- process_minout from **biobb_amber.process.process_minout**

```
# Import module
from biobb_amber.sander.sander_mdrun import sander_mdrun

# Create prop dict and inputs/outputs
prop = {
    "mdin" : {
        'maxcyc' : 500  # Overwrite number of minimization steps if needed
    },
#    "sander_path" : "sander.MPI",  # Change sander binary to parallel (MPI) execution␣
↪(not included in AmberTools)
#    "mpi_bin" : "mpirun",          # MPI runner
#    "mpi_np" : 16                  # Number of cores to use in the MPI parallel␣
↪calculation
}
output_eq5_traj_path = 'sander.eq5.nc'
output_eq5_rst_path = 'sander.eq5.ncrst'
output_eq5_log_path = 'sander.eq5.log'
output_eq5_mdinfo_path = 'sander.eq5.mdinfo'

# Create and launch bb
sander_mdrun(input_top_path=dna_leap_top_4fs_path,
             input_mdin_path="ABCix_config_files/step5.in",
             input_crd_path=output_eq4_rst_path,
             input_ref_path=output_eq4_rst_path,
             output_traj_path=output_eq5_traj_path,
             output_rst_path=output_eq5_rst_path,
             output_log_path=output_eq5_log_path,
             output_mdinfo_path=output_eq5_mdinfo_path,
             properties=prop)
```

Checking Equilibration Step 5 results

Checking **Equilibration Step 5 - System Energetic Minimization** results. Plotting **potential energy** along time during the **minimization process**.

```
# Import module
from biobb_amber.process.process_minout import process_minout

# Create prop dict and inputs/outputs
prop = {
    "terms" : ['ENERGY']
}
output_dat_eq5_path = 'sander.eq5.energy.dat'

# Create and launch bb
process_minout(input_log_path=output_eq5_log_path,
             output_dat_path=output_dat_eq5_path,
             properties=prop)
```

```
#Read data from file and filter energy values higher than 1000 Kj/mol^-1
with open(output_dat_eq5_path,'r') as energy_file:
```

(continues on next page)

(continued from previous page)

```
    x,y = map(
        list,
        zip(*[
            (float(line.split()[0]),float(line.split()[1]))
            for line in energy_file
            if not line.startswith(("#","@"))
            if float(line.split()[1]) < 1000
        ])
    )

plotly.offline.init_notebook_mode(connected=True)

fig = {
    "data": [go.Scatter(x=x, y=y)],
    "layout": go.Layout(title="Equilibration Step 5",
                        xaxis=dict(title = "Energy Minimization Step"),
                        yaxis=dict(title = "Potential Energy kcal/mol")
                       )
}

plotly.offline.iplot(fig)
```

Equilibration Step 6: NPT equilibration

Equilibrate the **system** in **NPT** ensemble (constant number of particles -N-, Pressure -P-, and Temperature -T-), using the **sander tool** from the **AMBER MD package**.

AMBER MD configuration file used (step6.in) includes the following **simulation parameters**:

- imin = 0; Run MD (no minimization)
- nstlim = 5000; Number of MD steps
- dt = 0.001; Time step (in ps)
- ntx = 1; Only read initial coordinates from input files
- irest = 0; Do not restart a simulation, start a new one
- ig = -1; Seed for the pseudo-random number generator
- ntwx = 500; Coordinates will be written to the output trajectory file every 500 steps
- ntwv = -1; Velocities will be written to output trajectory file, making it a combined coordinate/velocity trajectory file, at the interval defined by ntwx
- ioutfm = 1; Write trajectory in netcdf format
- ntxo = 2; Write restart in netcdf format
- ntpr = 50; Write energy information to files 'mdout' and 'mdinfo' every 50 steps
- ntwr = 500; Write information to restart file every 500 steps
- iwrap = 0; Not wrapping coordinates into primary box

- nscm = 0; Not removing translational and rotational center-of-mass motions
- ntc = 2; Turn on SHAKE for constraining length of bonds involving Hydrogen atoms
- ntf = 2; Force evaluation: Bond interactions involving H omitted (SHAKE)
- ntb = 2; Constant Pressure Periodic Boundary Conditions (PBC)
- cut = 8.0; Cutoff for non bonded interactions in Angstroms
- ntt = 3; Constant temperature using Langevin dynamics
- gamma_ln = 5; Collision frequency for Langevin dynamics (in 1/ps)
- temp0 = 310.0; Final temperature (310 K)
- tempi = 310.0; Initial temperature (310 K)
- ntp = 1; Constant pressure dynamics: md with isotropic position scaling
- barostat = 2; Monte Carlo Barostat
- pres0 = 1.0; Reference pressure at which the system is maintained
- ntr = 1; Turn on positional restraints
- restraintmask = :1-40&!@H=; Restraints on DNA atoms only
- restraint_wt = 1.0; Restraint force constant

NPT equilibration step applying **restraints** on the **DNA heavy atoms** with a **force constant** of **1 Kcal/mol.$Å^{2}$**

Building Blocks used:

- sander_mdrun from **biobb_amber.sander.sander_mdrun**
- process_mdout from **biobb_amber.process.process_mdout**

```
# Import module
from biobb_amber.sander.sander_mdrun import sander_mdrun

# Create prop dict and inputs/outputs
prop = {
    "mdin" : {
        'nstlim' : 500,  # Overwrite number of MD steps if needed
        'restraintmask' : '\":DA,DC,DG,DT,D=3,D=5&!@H=\"' # Overwrite DNA heavy atoms␣
↪mask to make it more generic
    },
#    "sander_path" : "sander.MPI",  # Change sander binary to parallel (MPI) execution␣
↪(not included in AmberTools)
#    "mpi_bin" : "mpirun",          # MPI runner
#    "mpi_np" : 16                  # Number of cores to use in the MPI parallel␣
↪calculation
}
output_eq6_traj_path = 'sander.eq6.nc'
output_eq6_rst_path = 'sander.eq6.ncrst'
output_eq6_log_path = 'sander.eq6.log'
output_eq6_mdinfo_path = 'sander.eq6.mdinfo'
```

(continues on next page)

(continued from previous page)

```
# Create and launch bb
sander_mdrun(input_top_path=dna_leap_top_4fs_path,
             input_mdin_path="ABCix_config_files/step6.in",
             input_crd_path=output_eq5_rst_path,
             input_ref_path=output_eq5_rst_path,
             output_traj_path=output_eq6_traj_path,
             output_rst_path=output_eq6_rst_path,
             output_log_path=output_eq6_log_path,
             output_mdinfo_path=output_eq6_mdinfo_path,
             properties=prop)
```

Checking Equilibration Step 6 results

Checking **Equilibration Step 6 - NPT Equilibration** results. Plotting **density** and **pressure** by time during the **equilibration process**.

```
# Import module
from biobb_amber.process.process_mdout import process_mdout

# Create prop dict and inputs/outputs
prop = {
    "terms" : ['PRES','DENSITY']
}
output_dat_eq6_path = 'sander.eq6.pressure_and_density.dat'

# Create and launch bb
process_mdout(input_log_path=output_eq6_log_path,
             output_dat_path=output_dat_eq6_path,
             properties=prop)
```

```
# Read pressure and density data from file
from plotly import subplots
with open(output_dat_eq6_path,'r') as pd_file:
    x,y,z = map(
        list,
        zip(*[
            (float(line.split()[0]),float(line.split()[1]),float(line.split()[2]))
            for line in pd_file
            if not line.startswith(("#","@"))
        ])
    )

plotly.offline.init_notebook_mode(connected=True)

trace1 = go.Scatter(
    x=x,y=y
)
trace2 = go.Scatter(
    x=x,y=z
)
```

(continues on next page)

(continued from previous page)

```
fig = subplots.make_subplots(rows=1, cols=2, print_grid=False)

fig.append_trace(trace1, 1, 1)
fig.append_trace(trace2, 1, 2)

fig['layout']['xaxis1'].update(title='Time (ps)')
fig['layout']['xaxis2'].update(title='Time (ps)')
fig['layout']['yaxis1'].update(title='Pressure (bar)')
fig['layout']['yaxis2'].update(title='Density (Kg*m^-3)')

fig['layout'].update(title='Pressure and Density during NPT Equilibration')
fig['layout'].update(showlegend=False)

plotly.offline.iplot(fig)
```

Equilibration Step 7: NPT equilibration

Equilibrate the **system** in **NPT** ensemble (constant number of particles -N-, Pressure -P-, and Temperature -T-), using the **sander tool** from the **AMBER MD package**. Lowering **restraints force constant**.

AMBER MD configuration file used (step7.in) includes the following **simulation parameters**:

- imin = 0; Run MD (no minimization)
- nstlim = 5000; Number of MD steps
- dt = 0.001; Time step (in ps)
- ntx = 5; Read initial coordinates and velocities from restart file
- irest = 1; Restart previous simulation from restart file
- ig = -1; Seed for the pseudo-random number generator
- ntwx = 500; Coordinates will be written to the output trajectory file every 500 steps
- ntwv = -1; Velocities will be written to output trajectory file, making it a combined coordinate/velocity trajectory file, at the interval defined by ntwx
- ioutfm = 1; Write trajectory in netcdf format
- ntxo = 2; Write restart in netcdf format
- ntpr = 50; Write energy information to files 'mdout' and 'mdinfo' every 50 steps
- ntwr = 500; Write information to restart file every 500 steps
- iwrap = 0; Not wrapping coordinates into primary box
- nscm = 0; Not removing translational and rotational center-of-mass motions
- ntc = 2; Turn on SHAKE for constraining length of bonds involving Hydrogen atoms
- ntf = 2; Force evaluation: Bond interactions involving H omitted (SHAKE)
- ntb = 2; Constant Pressure Periodic Boundary Conditions (PBC)
- cut = 8.0; Cutoff for non bonded interactions in Angstroms

- ntt = 3; Constant temperature using Langevin dynamics
- gamma_ln = 5; Collision frequency for Langevin dynamics (in 1/ps)
- temp0 = 310.0; Final temperature (310 K)
- tempi = 310.0; Initial temperature (310 K)
- ntp = 1; Constant pressure dynamics: md with isotropic position scaling
- barostat = 2; Monte Carlo Barostat
- pres0 = 1.0; Reference pressure at which the system is maintained
- ntr = 1; Turn on positional restraints
- restraintmask = :1-40&!@H=; Restraints on DNA atoms only
- restraint_wt = 0.5; Restraint force constant

NPT equilibration step applying **restraints** on the **DNA heavy atoms** with a **force constant** of **0.5 Kcal/mol.$Å^{2}$**. **Restarting** from previous equilibration step.

Building Blocks used:

- sander_mdrun from **biobb_amber.sander.sander_mdrun**
- process_mdout from **biobb_amber.process.process_mdout**

```
# Import module
from biobb_amber.sander.sander_mdrun import sander_mdrun

# Create prop dict and inputs/outputs
prop = {
    "mdin" : {
        'nstlim' : 500,  # Overwrite number of MD steps if needed
        'restraintmask' : '\":DA,DC,DG,DT,D=3,D=5&!@H=\"' # Overwrite DNA heavy atoms␣
↪mask to make it more generic
    },
#    "sander_path" : "sander.MPI",  # Change sander binary to parallel (MPI) execution␣
↪(not included in AmberTools)
#    "mpi_bin" : "mpirun",          # MPI runner
#    "mpi_np" : 16                  # Number of cores to use in the MPI parallel␣
↪calculation
}
output_eq7_traj_path = 'sander.eq7.nc'
output_eq7_rst_path = 'sander.eq7.ncrst'
output_eq7_log_path = 'sander.eq7.log'
output_eq7_mdinfo_path = 'sander.eq7.mdinfo'

# Create and launch bb
sander_mdrun(input_top_path=dna_leap_top_4fs_path,
             input_mdin_path="ABCix_config_files/step7.in",
             input_crd_path=output_eq6_rst_path,
             input_ref_path=output_eq6_rst_path,
             output_traj_path=output_eq7_traj_path,
             output_rst_path=output_eq7_rst_path,
```

(continues on next page)

(continued from previous page)

```
            output_log_path=output_eq7_log_path,
            output_mdinfo_path=output_eq7_mdinfo_path,
            properties=prop)
```

Checking Equilibration Step 7 results

Checking **Equilibration Step 7 - NPT Equilibration** results. Plotting **density** and **pressure** by time during the **equilibration process**.

```
# Import module
from biobb_amber.process.process_mdout import process_mdout

# Create prop dict and inputs/outputs
prop = {
    "terms" : ['PRES','DENSITY']
}
output_dat_eq7_path = 'sander.eq7.pressure_and_density.dat'

# Create and launch bb
process_mdout(input_log_path=output_eq7_log_path,
            output_dat_path=output_dat_eq7_path,
            properties=prop)
```

```
# Read pressure and density data from file
from plotly import subplots
with open(output_dat_eq7_path,'r') as pd_file:
    x,y,z = map(
        list,
        zip(*[
            (float(line.split()[0]),float(line.split()[1]),float(line.split()[2]))
            for line in pd_file
            if not line.startswith(("#","@"))
        ])
    )

plotly.offline.init_notebook_mode(connected=True)

trace1 = go.Scatter(
    x=x,y=y
)
trace2 = go.Scatter(
    x=x,y=z
)

fig = subplots.make_subplots(rows=1, cols=2, print_grid=False)

fig.append_trace(trace1, 1, 1)
fig.append_trace(trace2, 1, 2)

fig['layout']['xaxis1'].update(title='Time (ps)')
fig['layout']['xaxis2'].update(title='Time (ps)')
```

(continues on next page)

(continued from previous page)

```
fig['layout']['yaxis1'].update(title='Pressure (bar)')
fig['layout']['yaxis2'].update(title='Density (Kg*m^-3)')

fig['layout'].update(title='Pressure and Density during NPT Equilibration')
fig['layout'].update(showlegend=False)

plotly.offline.iplot(fig)
```

Equilibration Step 8: NPT equilibration

Equilibrate the **system** in **NPT** ensemble (constant number of particles -N-, Pressure -P-, and Temperature -T-), using the **sander tool** from the **AMBER MD package**. Releasing **position restraints** from the atoms of the **DNA bases** (keeping only backbone atoms restrained).

AMBER MD configuration file used (step8.in) includes the following **simulation parameters**:

- imin = 0; Run MD (no minimization)
- nstlim = 10000; Number of MD steps
- dt = 0.001; Time step (in ps)
- ntx = 5; Read initial coordinates and velocities from restart file
- irest = 1; Restart previous simulation from restart file
- ig = -1; Seed for the pseudo-random number generator
- ntwx = 500; Coordinates will be written to the output trajectory file every 500 steps
- ntwv = -1; Velocities will be written to output trajectory file, making it a combined coordinate/velocity trajectory file, at the interval defined by ntwx
- ioutfm = 1; Write trajectory in netcdf format
- ntxo = 2; Write restart in netcdf format
- ntpr = 50; Write energy information to files 'mdout' and 'mdinfo' every 50 steps
- ntwr = 500; Write information to restart file every 500 steps
- iwrap = 0; Not wrapping coordinates into primary box
- nscm = 0; Not removing translational and rotational center-of-mass motions
- ntc = 2; Turn on SHAKE for constraining length of bonds involving Hydrogen atoms
- ntf = 2; Force evaluation: Bond interactions involving H omitted (SHAKE)
- ntb = 2; Constant Pressure Periodic Boundary Conditions (PBC)
- cut = 8.0; Cutoff for non bonded interactions in Angstroms
- ntt = 3; Constant temperature using Langevin dynamics
- gamma_ln = 5; Collision frequency for Langevin dynamics (in 1/ps)
- temp0 = 310.0; Final temperature (310 K)
- tempi = 310.0; Initial temperature (310 K)

- ntp = 1; Constant pressure dynamics: md with isotropic position scaling
- barostat = 2; Monte Carlo Barostat
- pres0 = 1.0; Reference pressure at which the system is maintained
- ntr = 1; Turn on positional restraints
- restraintmask = :1-40@P,O5',C5',C4',C3',O3'; Restraints on DNA atoms only
- restraint_wt = 0.5; Restraint force constant

NPT equilibration step applying **restraints** on the **DNA backbone atoms** with a **force constant** of **0.5 Kcal/mol.$Å^{2}$**. **Restarting** from previous equilibration step.

Building Blocks used:

- sander_mdrun from **biobb_amber.sander.sander_mdrun**
- process_mdout from **biobb_amber.process.process_mdout**

```
# Import module
from biobb_amber.sander.sander_mdrun import sander_mdrun

# Create prop dict and inputs/outputs
prop = {
    "mdin" : {
        'nstlim' : 500,  # Overwrite number of MD steps if needed
        'restraintmask' : '\":DA,DC,DG,DT,D=3,D=5&@P,O5\',C5\',C4\',C3\',O3\'\"' #␣
↪Overwrite DNA heavy atoms mask to make it more generic
    },
#    "sander_path" : "sander.MPI",  # Change sander binary to parallel (MPI) execution␣
↪(not included in AmberTools)
#    "mpi_bin" : "mpirun",          # MPI runner
#    "mpi_np" : 16                  # Number of cores to use in the MPI parallel␣
↪calculation
}
output_eq8_traj_path = 'sander.eq8.nc'
output_eq8_rst_path = 'sander.eq8.ncrst'
output_eq8_log_path = 'sander.eq8.log'
output_eq8_mdinfo_path = 'sander.eq8.mdinfo'

# Create and launch bb
sander_mdrun(input_top_path=dna_leap_top_4fs_path,
             input_mdin_path="ABCix_config_files/step8.in",
             input_crd_path=output_eq7_rst_path,
             input_ref_path=output_eq7_rst_path,
             output_traj_path=output_eq8_traj_path,
             output_rst_path=output_eq8_rst_path,
             output_log_path=output_eq8_log_path,
             output_mdinfo_path=output_eq8_mdinfo_path,
             properties=prop)
```

Checking Equilibration Step 8 results

Checking **Equilibration Step 8 - NPT Equilibration** results. Plotting **density** and **pressure** by time during the **equilibration process**.

```
# Import module
from biobb_amber.process.process_mdout import process_mdout

# Create prop dict and inputs/outputs
prop = {
    "terms" : ['PRES','DENSITY']
}
output_dat_eq8_path = 'sander.eq8.pressure_and_density.dat'

# Create and launch bb
process_mdout(input_log_path=output_eq8_log_path,
            output_dat_path=output_dat_eq8_path,
            properties=prop)
```

```
# Read pressure and density data from file
from plotly import subplots
with open(output_dat_eq8_path,'r') as pd_file:
    x,y,z = map(
        list,
        zip(*[
            (float(line.split()[0]),float(line.split()[1]),float(line.split()[2]))
            for line in pd_file
            if not line.startswith(("#","@"))
        ])
    )

plotly.offline.init_notebook_mode(connected=True)

trace1 = go.Scatter(
    x=x,y=y
)
trace2 = go.Scatter(
    x=x,y=z
)

fig = subplots.make_subplots(rows=1, cols=2, print_grid=False)

fig.append_trace(trace1, 1, 1)
fig.append_trace(trace2, 1, 2)

fig['layout']['xaxis1'].update(title='Time (ps)')
fig['layout']['xaxis2'].update(title='Time (ps)')
fig['layout']['yaxis1'].update(title='Pressure (bar)')
fig['layout']['yaxis2'].update(title='Density (Kg*m^-3)')

fig['layout'].update(title='Pressure and Density during NPT Equilibration')
fig['layout'].update(showlegend=False)
```

(continues on next page)

(continued from previous page)

```
plotly.offline.iplot(fig)
```

Equilibration Step 9: NPT equilibration

Equilibrate the **system** in **NPT** ensemble (constant number of particles -N-, Pressure -P-, and Temperature -T-), using the **sander tool** from the **AMBER MD package**. Releasing all **position restraints**.

AMBER MD configuration file used (step9.in) includes the following **simulation parameters**:

- imin = 0; Run MD (no minimization)
- nstlim = 5000; Number of MD steps
- dt = 0.002; Time step (in ps)
- ntx = 5; Read initial coordinates and velocities from restart file
- irest = 1; Restart previous simulation from restart file
- ig = -1; Seed for the pseudo-random number generator
- ntwx = 500; Coordinates will be written to the output trajectory file every 500 steps
- ntwv = -1; Velocities will be written to output trajectory file, making it a combined coordinate/velocity trajectory file, at the interval defined by ntwx
- ioutfm = 1; Write trajectory in netcdf format
- ntxo = 2; Write restart in netcdf format
- ntpr = 50; Write energy information to files 'mdout' and 'mdinfo' every 50 steps
- ntwr = 500; Write information to restart file every 500 steps
- iwrap = 0; Not wrapping coordinates into primary box
- nscm = 1000; Removing translational and rotational center-of-mass motions every 1000 steps
- ntc = 2; Turn on SHAKE for constraining length of bonds involving Hydrogen atoms
- ntf = 2; Force evaluation: Bond interactions involving H omitted (SHAKE)
- ntb = 2; Constant Pressure Periodic Boundary Conditions (PBC)
- cut = 8.0; Cutoff for non bonded interactions in Angstroms
- ntt = 3; Constant temperature using Langevin dynamics
- gamma_ln = 5; Collision frequency for Langevin dynamics (in 1/ps)
- temp0 = 310.0; Final temperature (310 K)
- tempi = 310.0; Initial temperature (310 K)
- ntp = 1; Constant pressure dynamics: md with isotropic position scaling
- barostat = 2; Monte Carlo Barostat
- pres0 = 1.0; Reference pressure at which the system is maintained
- ntr = 0; Turn off positional restraints

NPT equilibration step without any **position restraints**. Removing **translational and rotational** center-of-mass motions. **Timestep** 2fs. **Restarting** from previous equilibration step.

Building Blocks used:

- sander_mdrun from **biobb_amber.sander.sander_mdrun**
- process_mdout from **biobb_amber.process.process_mdout**

```
# Import module
from biobb_amber.sander.sander_mdrun import sander_mdrun

# Create prop dict and inputs/outputs
prop = {
    "mdin" : {
        'nstlim' : 500  # Overwrite number of MD steps if needed
    },
#    "sander_path" : "sander.MPI",  # Change sander binary to parallel (MPI) execution␣
↪(not included in AmberTools)
#    "mpi_bin" : "mpirun",          # MPI runner
#    "mpi_np" : 16                  # Number of cores to use in the MPI parallel␣
↪calculation
}
output_eq9_traj_path = 'sander.eq9.nc'
output_eq9_rst_path = 'sander.eq9.ncrst'
output_eq9_log_path = 'sander.eq9.log'
output_eq9_mdinfo_path = 'sander.eq9.mdinfo'

# Create and launch bb
sander_mdrun(input_top_path=dna_leap_top_4fs_path,
             input_mdin_path="ABCix_config_files/step9.in",
             input_crd_path=output_eq8_rst_path,
             input_ref_path=output_eq8_rst_path,
             output_traj_path=output_eq9_traj_path,
             output_rst_path=output_eq9_rst_path,
             output_log_path=output_eq9_log_path,
             output_mdinfo_path=output_eq9_mdinfo_path,
             properties=prop)
```

Checking Equilibration Step 9 results

Checking **Equilibration Step 9 - NPT Equilibration** results. Plotting **density** and **pressure** by time during the **equilibration process**.

```
# Import module
from biobb_amber.process.process_mdout import process_mdout

# Create prop dict and inputs/outputs
prop = {
    "terms" : ['PRES','DENSITY']
}
```

(continues on next page)

(continued from previous page)

```
output_dat_eq9_path = 'sander.eq9.pressure_and_density.dat'

# Create and launch bb
process_mdout(input_log_path=output_eq9_log_path,
              output_dat_path=output_dat_eq9_path,
              properties=prop)
```

```
# Read pressure and density data from file
from plotly import subplots
with open(output_dat_eq9_path,'r') as pd_file:
    x,y,z = map(
        list,
        zip(*[
            (float(line.split()[0]),float(line.split()[1]),float(line.split()[2]))
            for line in pd_file
            if not line.startswith(("#","@"))
        ])
    )

plotly.offline.init_notebook_mode(connected=True)

trace1 = go.Scatter(
    x=x,y=y
)
trace2 = go.Scatter(
    x=x,y=z
)

fig = subplots.make_subplots(rows=1, cols=2, print_grid=False)

fig.append_trace(trace1, 1, 1)
fig.append_trace(trace2, 1, 2)

fig['layout']['xaxis1'].update(title='Time (ps)')
fig['layout']['xaxis2'].update(title='Time (ps)')
fig['layout']['yaxis1'].update(title='Pressure (bar)')
fig['layout']['yaxis2'].update(title='Density (Kg*m^-3)')

fig['layout'].update(title='Pressure and Density during NPT Equilibration')
fig['layout'].update(showlegend=False)

plotly.offline.iplot(fig)
```

Equilibration Step 10: NPT equilibration

Equilibrate the **system** in **NPT** ensemble (constant number of particles -N-, Pressure -P-, and Temperature -T-), using the **sander tool** from the **AMBER MD package**. Upon completion of the previous **equilibration steps**, the system is now well-equilibrated at the desired **temperature** and **pressure**. The last step of the **DNA** MD setup is a short, **free MD simulation**, to ensure the robustness of the system.

AMBER MD configuration file used (step10.in) includes the following **simulation parameters**:

- imin = 0; Run MD (no minimization)
- nstlim = 500000; Number of MD steps
- dt = 0.002; Time step (in ps)
- ntx = 5; Read initial coordinates and velocities from restart file
- irest = 1; Restart previous simulation from restart file
- ig = -1; Seed for the pseudo-random number generator
- ntwx = 5000; Coordinates will be written to the output trajectory file every 5000 steps
- ntwv = -1; Velocities will be written to output trajectory file, making it a combined coordinate/velocity trajectory file, at the interval defined by ntwx
- ioutfm = 1; Write trajectory in netcdf format
- ntxo = 2; Write restart in netcdf format
- ntpr = 500; Write energy information to files 'mdout' and 'mdinfo' every 500 steps
- ntwr = 50000; Write information to restart file every 50000 steps
- iwrap = 0; Not wrapping coordinates into primary box
- nscm = 1000; Removing translational and rotational center-of-mass motions every 1000 steps
- ntc = 2; Turn on SHAKE for constraining length of bonds involving Hydrogen atoms
- ntf = 2; Force evaluation: Bond interactions involving H omitted (SHAKE)
- ntb = 2; Constant Pressure Periodic Boundary Conditions (PBC)
- cut = 9.0; Cutoff for non bonded interactions in Angstroms
- ntt = 3; Constant temperature using Langevin dynamics
- gamma_ln = 5; Collision frequency for Langevin dynamics (in 1/ps)
- temp0 = 310.0; Final temperature (310 K)
- tempi = 310.0; Initial temperature (310 K)
- ntp = 1; Constant pressure dynamics: md with isotropic position scaling
- barostat = 2; Monte Carlo Barostat
- pres0 = 1.0; Reference pressure at which the system is maintained
- ntr = 0; Turn off positional restraints

NPT equilibration step without any **position restraints**. Removing **translational and rotational** center-of-mass motions. **Timestep** 2fs. **Writing times** changed (production MD). **Restarting** from previous equilibration step.

Building Blocks used:

- sander_mdrun from **biobb_amber.sander.sander_mdrun**
- process_mdout from **biobb_amber.process.process_mdout**

```
# Import module
from biobb_amber.sander.sander_mdrun import sander_mdrun

# Create prop dict and inputs/outputs
prop = {
    "mdin" : {
        'nstlim' : 500,  # Overwrite number of MD steps if needed
        'ntpr' : 50      # Overwrite energy information writing frequency
    },
#    "sander_path" : "sander.MPI",  # Change sander binary to parallel (MPI) execution␣
↪(not included in AmberTools)
#    "mpi_bin" : "mpirun",          # MPI runner
#    "mpi_np" : 16                  # Number of cores to use in the MPI parallel␣
↪calculation
}
output_eq10_traj_path = 'sander.eq10.nc'
output_eq10_rst_path = 'sander.eq10.ncrst'
output_eq10_log_path = 'sander.eq10.log'
output_eq10_mdinfo_path = 'sander.eq10.mdinfo'

# Create and launch bb
sander_mdrun(input_top_path=dna_leap_top_4fs_path,
             input_mdin_path="ABCix_config_files/step10.in",
             input_crd_path=output_eq9_rst_path,
             input_ref_path=output_eq9_rst_path,
             output_traj_path=output_eq10_traj_path,
             output_rst_path=output_eq10_rst_path,
             output_log_path=output_eq10_log_path,
             output_mdinfo_path=output_eq10_mdinfo_path,
             properties=prop)
```

Checking Equilibration Step 10 results

Checking **Equilibration Step 10 - NPT Equilibration** results. Plotting **density** and **pressure** by time during the **equilibration process**.

```
# Import module
from biobb_amber.process.process_mdout import process_mdout

# Create prop dict and inputs/outputs
prop = {
    "terms" : ['PRES','DENSITY']
}
output_dat_eq10_path = 'sander.eq10.pressure_and_density.dat'

# Create and launch bb
process_mdout(input_log_path=output_eq10_log_path,
```

(continues on next page)

(continued from previous page)

```
            output_dat_path=output_dat_eq10_path,
            properties=prop)
```

```
# Read pressure and density data from file
from plotly import subplots
with open(output_dat_eq10_path,'r') as pd_file:
    x,y,z = map(
        list,
        zip(*[
            (float(line.split()[0]),float(line.split()[1]),float(line.split()[2]))
            for line in pd_file
            if not line.startswith(("#","@"))
        ])
    )

plotly.offline.init_notebook_mode(connected=True)

trace1 = go.Scatter(
    x=x,y=y
)
trace2 = go.Scatter(
    x=x,y=z
)

fig = subplots.make_subplots(rows=1, cols=2, print_grid=False)

fig.append_trace(trace1, 1, 1)
fig.append_trace(trace2, 1, 2)

fig['layout']['xaxis1'].update(title='Time (ps)')
fig['layout']['xaxis2'].update(title='Time (ps)')
fig['layout']['yaxis1'].update(title='Pressure (bar)')
fig['layout']['yaxis2'].update(title='Density (Kg*m^-3)')

fig['layout'].update(title='Pressure and Density during NPT Equilibration')
fig['layout'].update(showlegend=False)

plotly.offline.iplot(fig)
```

1.5.12 Free Molecular Dynamics Simulation

Upon completion of the **10 equilibration phases**, the system can now be used for the production simulation. In the **ABCix** run of the **ABC consortium**, a **4fs timestep** is used, applying **hydrogen-mass repartition**. **Langevin** algorithm is used for the **temperature coupling** and **Monte Carlo barostat** algorithm for **pressure coupling**.

AMBER MD configuration file used (md.in) includes the following **simulation parameters**:

- imin = 0; Run MD (no minimization)
- nstlim = 500000; Number of MD steps – 1ns
- dt = 0.004; Time step (in ps) – using H-mass repartition
- ntx = 5; Read initial coordinates and velocities from restart file
- irest = 1; Restart previous simulation from restart file
- ig = -1; Seed for the pseudo-random number generator
- ntpr = 5000; Write energy information to files 'mdout' and 'mdinfo' every 5000 steps – 20ps
- ntwr = 5000; Write information to restart file every 5000 steps – 20ps
- ntwx = 5000; Coordinates will be written to the output trajectory file every 5000 steps – 20ps
- ntc = 2; Turn on SHAKE for constraining length of bonds involving Hydrogen atoms
- ntf = 2; Force evaluation: Bond interactions involving H omitted (SHAKE)
- iwrap = 1; Wrapping coordinates into primary box
- ntb = 2; Constant Pressure Periodic Boundary Conditions (PBC)
- ntp = 1; Constant pressure dynamics: md with isotropic position scaling
- barostat = 2; Monte Carlo Barostat
- mcbarint = 100; Number of steps between volume change attempts performed as part of the Monte Carlo barostat
- pres0 = 1.0; Reference pressure at which the system is maintained
- cut = 8.0; Cutoff for non bonded interactions in Angstroms
- temp0 = 310.0; Final temperature (310 K)
- ntt = 3; Constant temperature using Langevin dynamics
- gamma_ln = 0.01; Collision frequency for Langevin dynamics (in 1/ps)
- tol = 0.0000001; Relative geometrical tolerance for coordinate resetting in shake
- timlim = 170000; Time limit for the simulation

Note: *Although not stated explicitly, all output files will be generated in binary netcdf format (default for ioutfm and ntxo parameters).*

Building Blocks used:

- sander_mdrun from **biobb_amber.sander.sander_mdrun**
- process_mdout from **biobb_amber.process.process_mdout**

```
# Import module
from biobb_amber.sander.sander_mdrun import sander_mdrun

# Create prop dict and inputs/outputs
prop = {
    "mdin" : {
        'nstlim' : 500,  # Overwrite number of MD steps if needed
        'ntpr' : 50      # Overwrite energy information writing frequency
    },
#    "sander_path" : "sander.MPI",  # Change sander binary to parallel (MPI) execution␣
↪(not included in AmberTools)
#    "mpi_bin" : "mpirun",          # MPI runner
#    "mpi_np" : 16                  # Number of cores to use in the MPI parallel␣
↪calculation
}
output_md_traj_path = 'sander.md.nc'
output_md_rst_path = 'sander.md.ncrst'
output_md_log_path = 'sander.md.log'
output_md_mdinfo_path = 'sander.md.mdinfo'

# Create and launch bb
sander_mdrun(input_top_path=dna_leap_top_4fs_path,
             input_mdin_path="ABCix_config_files/md.in",
             input_crd_path=output_eq10_rst_path,
             input_ref_path=output_eq10_rst_path,
             output_traj_path=output_md_traj_path,
             output_rst_path=output_md_rst_path,
             output_log_path=output_md_log_path,
             output_mdinfo_path=output_md_mdinfo_path,
             properties=prop)
```

1.5.13 Output files

Important **Output files** generated:

- sander.md.nc: **Final trajectory** of the MD setup protocol (netcdf).
- structure.ions.parmtop: **Final topology** of the MD system.
- structure.leap.4fs.top: **Final topology** of the MD system with **hydrogen mass repartition** (allowing 4fs timestep).
- sander.md.ncrst: **Final restart file** of the MD setup protocol (ncrst).

```
from IPython.display import FileLink
display(FileLink(output_md_traj_path))
display(FileLink(output_ions_top_path))
display(FileLink(dna_leap_top_4fs_path))
display(FileLink(output_md_rst_path))
```

sander.md.nc

structure.ions.parmtop

structure.leap.4fs.top

sander.md.ncrst

1.5.14 Questions & Comments

Questions, issues, suggestions and comments are really welcome!

- GitHub issues:
  - https://github.com/bioexcel/biobb
- BioExcel forum:
  - https://ask.bioexcel.eu/c/BioExcel-Building-Blocks-library

CHAPTER

TWO

GITHUB REPOSITORY.