mbuild

Release 0.17.1

Mosdef Team

May 10, 2024

Getting Started

`license` `MIT` [1] *A hierarchical, component based molecule builder*

With just a few lines of mBuild code, you can assemble reusable components into complex molecular systems for molecular simulations.

- mBuild is designed to minimize or even eliminate the need to explicitly translate and orient components when building systems: you simply tell it to connect two pieces!

- mBuild keeps track of the system's topology so you don't have to worry about manually defining bonds when constructing chemically bonded structures from smaller components.

---

[1] http://opensource.org/licenses/MIT

1 mBuild is a part of the MoSDeF ecosystem

The **mBuild** software, in conjunction with the other Molecular Simulation Design Framework (MoS-DeF)[2] tools, supports a wide range of simulation engines, including Cassandra[3], GPU Optimized Monte Carlo (GOMC)[4], GROMACS[5], HOOMD-blue[6], and Large-scale Atomic/Molecular Massively Parallel Simulator (LAMMPS)[7]. The **mBuild** and **MoSDeF** tools allow simulation reproducibility across the various simulation engines, eliminating the need to be an expert user in all the engines to replicate, continue, or advance the existing research. Additionally, the software can auto-generate many different systems, allowing large-scale screening of chemicals and materials using Signac[8] to manage the simulations and data.

The MoSDeF[9] software is comprised the following packages:

- mBuild[10] – A hierarchical, component based molecule builder

- foyer[11] – A package for atom-typing as well as applying and disseminating forcefields

- GMSO[12] – Flexible storage of chemical topology for molecular simulation

1.1 Example System

Components in dashed boxes are drawn by hand using, e.g., Avogadro[13] or generated elsewhere. mBuild[14] builds up complex systems from simple building blocks through simple attachment sites, called a `Port` (i.e., connection points). Each building block is a python class that can be customized or created through the pre-built options in the `mBuild` library (`mbuild.lib`). A hierarchical structure of parents and children is created through these classes, which can be easily parsed or modified. This allows mBuild[15] to generate chemical structures in a piecemeal fashion by creating or importing molecular sections, adding ports, and connecting the ports to form bonds. Together with Signac[16], this functionality enables an automatic and dynamic method for generating chemical systems, allowing large-scale chemical and materials screening with minimal user interaction.

Ultimately, complex systems can be created with just a line or two of code. Additionally, this approach seamlessly exposes tunable parameters within the hierarchy so you can actually create whole families of structures by adjusting a variable or two:

```
pattern = Random2DPattern(20)   # A random arrangement of 20 pieces on a 2D␣
↪surface.
brush_layer = BrushLayer(chain_lenth=20, pattern=pattern, tile_x=3, tile_y=2)
```

`license MIT` [17] Various sub-portions of this library may be independently distributed under different licenses. See those files for their specific terms.

---

[2] https://mosdef.org
[3] https://cassandra.nd.edu
[4] http://gomc.eng.wayne.edu
[5] https://www.gromacs.org
[6] http://glotzerlab.engin.umich.edu/hoomd-blue/
[7] https://lammps.sandia.gov
[8] https://signac.io
[9] https://mosdef.org
[10] https://mbuild.mosdef.org/en/stable/
[11] https://foyer.mosdef.org/en/stable/
[12] https://gmso.mosdef.org/en/stable/
[13] https://avogadro.cc
[14] https://mbuild.mosdef.org/en/stable/
[15] https://mbuild.mosdef.org/en/stable/
[16] https://signac.io
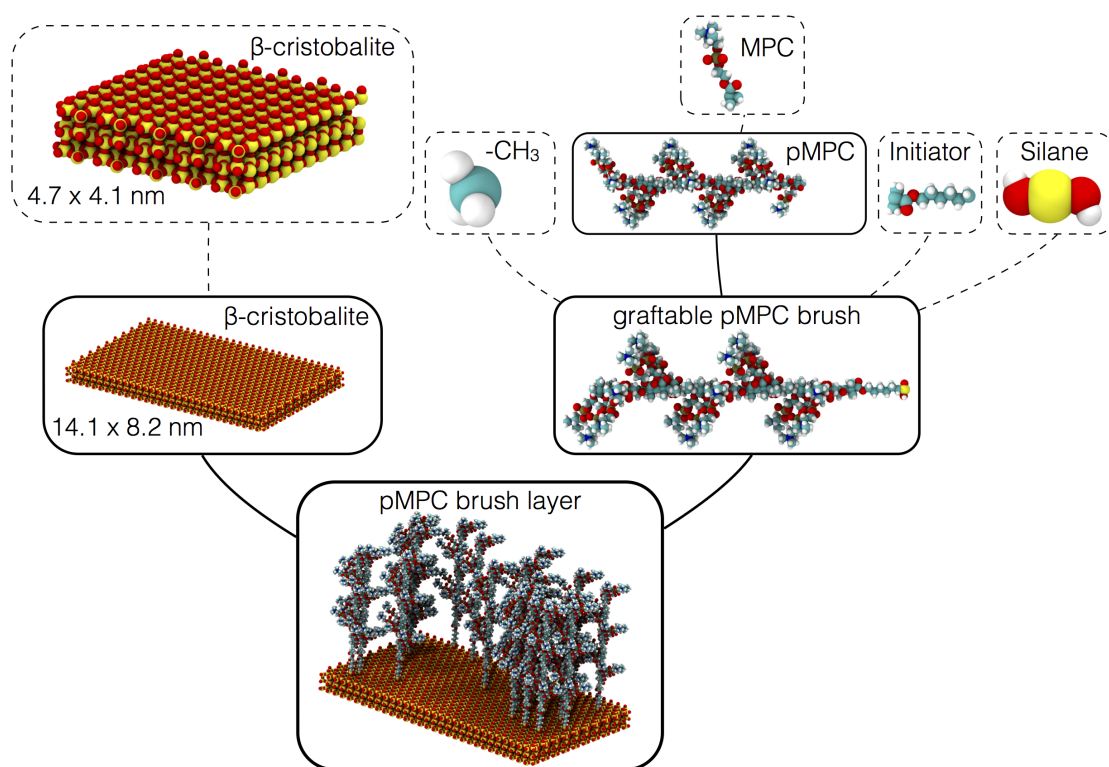[17] http://opensource.org/licenses/MIT

Fig. 1: **Zwitterionic brushes on beta-cristobalite substrate.** Example system that can be created using mBuild. Components in dashed boxes are created from some external tool like Avogadro or SMILES strings. Components in solid boxes are created from these smaller dashed components and then constructed into larger, more complex systems using mBuild functionality.

1.2 Installation

Installation

Install with conda[Page 4, 18]

```
$ conda install -c conda-forge mbuild
```

Alternatively you can add all the required channels to your `.condarc` after which you can simply install without specifying the channels:

```
$ conda config --add channels conda-forge
$ conda install mbuild
```

Note: The order in which channels are added matters: `conda-forge` should be the highest priority as a result of being added last. In your `.condarc` file, it should be listed first.

Note: Because `packmol` binaries are unavailable for windows from `conda-forge` channel, to use mbuild with conda in a Windows system requires the omnia channel. Use the following command to use `mbuild` with conda in a Windows system:

```
$ conda install -c conda-forge -c omnia mbuild
```

Note: The MDTraj website[19] makes a nice case for using Python and in particular the Anaconda scientific python distribution[20] to manage your numerical and scientific Python packages.

Install an editable version from source

To make your life easier, we recommend that you use a pre-packaged Python distribution like Miniconda[21] in order to get all of the dependencies:

```
$ git clone https://github.com/mosdef-hub/mbuild
$ cd mbuild
$ conda env create -f environment-dev.yml
$ conda activate mbuild-dev
$ pip install -e .
```

Note: The above installation is for OSX and Unix. If you are using Windows, use environment-win.yml instead of environment-dev.yml

---

[18] https://repo.anaconda.com/miniconda/
[19] http://mdtraj.org/1.9.3/new_to_python.html
[20] https://www.anaconda.com/products/individual
[21] https://docs.conda.io/en/latest/miniconda.html

Install pre-commit

We use pre-commit[22] to automatically handle our code formatting and this package is included in the dev environment. With the `mbuild-dev` conda environment active, pre-commit can be installed locally as a git hook by running:

```
$ pre-commit install
```

And (optional) all files can be checked by running:

```
$ pre-commit run --all-files
```

Supported Python Versions

Python 3.9, 3.10 and 3.11 are officially supported, including testing during development and packaging. Support for Python 2.7 has been dropped as of August 6, 2019. Other Python versions, such as 3.12 and 3.8 and older, may successfully build and function but no guarantee is made.

Testing your installation

mBuild uses py.test[23] for unit testing. To run them simply run the following while in the base directory:

```
$ conda install pytest
$ py.test -v
```

Building the documentation

mBuild uses sphinx[24] to build its documentation. To build the docs locally, run the following while in the `docs` directory:

```
$ cd docs
$ conda env create -f docs-env.yml
$ conda activate mbuild-docs
$ make html
```

Using mBuild with Docker

Docker and other containerization technologies allow entire applications and their dependencies to be packaged and distributed as images. This simplifies the installation process for the user and substantially reduces platform dependence (e.g., different compiler versions, libraries, etc). This section is a how-to guide for using mBuild with docker.

---

[22] https://pre-commit.com/
[23] https://docs.pytest.org/en/stable/
[24] https://www.sphinx-doc.org/en/master/index.html

Prerequisites

A docker installation on your machine. This Docker installation documentation[25] has instructions to get docker running on your machine. If you are not familiar with docker, the Internet is full of good tutorials like these from Docker curriculum[26] and YouTube[27].

Jupyter Quick Start

After you have a working docker installation, use the following command to start a Jupyter notebook with mBuild and all the required dependencies:

```
$ docker pull mosdef/mbuild:latest
$ docker run -it --name mbuild -p 8888:8888 mosdef/mbuild:latest
```

If no command is provided to the container (as above), the container starts a `jupyter-notebook` at the (container) location `/home/anaconda/data`. To access the notebook, paste the notebook URL into a web browser on your computer. When you are finished, you can use control-C to exit the notebook as usual. The docker container will exit upon notebook shutdown.

> **Warning:** Containers by nature are ephemeral, so filesystem changes (e.g., adding a new notebook) only persists until the end of the container's lifecycle. If the container is removed, any changes or code additions will not persist. See the section below for persistent data.

> **Note:** The `-it` flags connect your keyboard to the terminal running in the container. You may run the prior command without those flags, but be aware that the container will not respond to any keyboard input. In that case, you would need to use the `docker ps` and `docker kill` commands to shut down the container.

Persisting User Volumes

If you are using mBuild from a docker container and need access to data on your local machine or you wish to save files generated in the container, you can mount user volumes in the container. User volumes will provide a way to persist filesystem changes made to a container regardless of the container lifecycle. For example, you might want to create a directory called `mbuild-notebooks` on your local system, which will store all of your mBuild notebooks/code. In order to make that accessible from within the container (where the notebooks will be created/edited), use the following steps:

```
$ mkdir mbuild-notebooks
$ cd mbuild-notebooks/
$ docker run -it --name mbuild --mount type=bind,source=$(pwd),target=/home/
↪anaconda/data -p 8888:8888  mosdef/mbuild:latest
```

You can easily mount a different directory from your local machine by changing `source=$(pwd)` to `source=/path/to/my/favorite/directory`.

---

[25] https://docs.docker.com/get-docker/
[26] https://docker-curriculum.com/
[27] https://www.youtube.com/watch?v=zJ6WbK9zFpI&feature=youtu.be

Note: The `--mount` flag mounts a volume into the docker container. Here we use a `bind` mount to bind the current directory on our local filesystem to the `/home/anaconda/data` location in the container. The files you see in the `jupyter-notebook` browser window are those that exist on your local machine.

Warning: If you are using the container with jupyter notebooks you should use the `/home/anaconda/data` location as the mount point inside the container; this is the default notebook directory.

Running Python scripts in the container

Jupyter notebooks are a great way to explore new software and prototype code. However, when it comes time for production science, it is often better to work with python scripts. In order to execute a python script (`example.py`) that exists in the current working directory of your local machine, run:

```
$ docker run --mount type=bind,source=$(pwd),target=/home/anaconda/data mosdef/
↪mbuild:latest "python data/test.py"
```

Note that once again we are `bind` mounting the current working directory to `/home/anaconda/data`. The command we pass to the container is `python data/test.py`. Note the prefix `data/` to the script; this is because we enter the container in the home folder (`/home/anaconda`), but our script is located under `/home/anaconda/data`.

Warning: Do not bind mount to `target=/home/anaconda`. This will cause errors.

If you don't require a Jupyter notebook, but just want a Python interpreter, you can run:

```
$ docker run --mount type=bind,source=$(pwd),target=/home/anaconda/data mosdef/
↪mbuild:latest python
```

If you don't need access to any local data, you can of course drop the `--mount` command:

```
$ docker run mosdef/mbuild:latest python
```

Different mBuild versions

Instead of using `latest`, you can use the image `mosdef/mbuild:stable` for most recent stable release of mBuild.

Cleaning Up

You can remove the *container* by using the following command.

```
$ docker container rm mbuild
```

The *image* will still exist on your machine. See the tutorials at the top of this page for more information.

> **Warning:** You will not be able to start a second container with the same name (e.g., mbuild), until the first container has been removed.

Note: You do not need to name the container *mbuild* as shown in the above examples (`--name mbuild`). Docker will give each container a name automatically. To see all the containers on your machine, run `docker ps -a`.

1.3 Quick Start

license MIT [28]

The MoSDeF[Page 8, 29] software is comprised the following packages:

- mBuild[30] – A hierarchical, component based molecule builder
- foyer[31] – A package for atom-typing as well as applying and disseminating forcefields
- GMSO[32] – Flexible storage of chemical topology for molecular simulation

---

Note: foyer and **GMSO** are used together with **mBuild** to create all the required files to conduct the simulations. Run time parameters for a simulation engine need to be created by the user.

In the following examples, different types of simulation boxes are constructed using the **MoSDeF** software.

Molecular simulations are usually comprised of many molecules contained in a box (NPT and NVT ensembles), or boxes (GEMC and GCMC ensembles). The **mBuild** library allows for easy generation of the simulation box or boxes utilizing only a few lines of python code.

The following tutorials are available either as html or interactive jupyter[33] notebooks.

---

[28] http://opensource.org/licenses/MIT
[29] https://mosdef.org
[30] https://mbuild.mosdef.org/en/stable/
[31] https://foyer.mosdef.org/en/stable/
[32] https://gmso.mosdef.org/en/stable/
[33] https://jupyter.org/

Load files

mol2 files

Create an `mbuild.Compound` (i.e., the "pentane" variable) by loading a molecule from a mol2[34] file.

Import the required mbuild packages.

```python
import mbuild as mb
```

Load the "pentane.mol2" file from its directory.

```python
pentane = mb.load("path_to_mol2_file/pentane.mol2")
```

CIF files

Build an `mbuild.Compound` (i.e., the "ETV_triclinic" variable) by loading a Crystallographic Information File (CIF)[35] file and selecting the number of cell units to populate in the x, y, and z-dimensions.

Import the required mbuild packages.

```python
import mbuild as mb
from mbuild.lattice import load_cif
```

The CIF[36] file is loaded using the `load_cif` function. Next, three (3) cell units shall be built for all the x, y, and z-dimensions with the populate function. Finally, the CIF[37]'s residues are named 'ETV'.

```python
lattice_cif_ETV_triclinic = load_cif("path_to_cif_file/ETV_triclinic.cif")
ETV_triclinic = lattice_cif_ETV_triclinic.populate(x=3, y=3, z=3)
ETV_triclinic.name = 'ETV'
```

Other file types

mBuild also supports *Loading Data* or files via hoomd_snapshot, GSD, SMILES strings, and ParmEd structures.

Box

Import the required mbuild package.

```python
import mbuild as mb
```

---

[34] http://chemyang.ccnu.edu.cn/ccb/server/AIMMS/mol2.pdf
[35] https://www.iucr.org/resources/cif
[36] https://www.iucr.org/resources/cif
[37] https://www.iucr.org/resources/cif

Orthogonal Box

Build an empty orthogonal **mBuild** Box (i.e., the angle in degrees are = 90, = 90, = 90) measuring 4.0 nm in all the x, y, and z-dimensions.

Note: Note: if the angles are not specified, the system will default to an orthogonal box (i.e., the angle in degrees are = 90, = 90, = 90).

```
empty_box = mb.Box(lengths=[4.0, 4.0, 4.0], angles=[90, 90, 90])
```

Non-Orthogonal Box

Build an empty non-orthogonal **mBuild** Box (i.e., the angle in degrees are = 90, = 90, = 120) measuring 4.0 nm in the x and y-dimensions, and 5.0 nm in the z-dimension.

```
empty_box = mb.Box(lengths=[4.0, 4.0, 5.0], angles=[90, 90, 120])
```

Fill Box

All-Atom (AA) Hexane and Ethanol System

Note: foyer[38] is used in conjunction with mBuild in the following example to demonstrate how the MoSDeF[39] libraries can be used together to generate a simulation box.

Import the required mbuild package.

```
import mbuild as mb
```

Construct an all-atom (AA) hexane and ethanol using the OPLS-AA force field (FF), which is shipped as a standard foyer[40] FF. The hexane and ethanol molecules will be created using smiles strings[41]. The hexane and ethanol residues will be named *"HEX"* and *"ETO"*, respectively. Lastly, the hexane and ethanol molecule's configuration will be energy minimized, properly reorienting the molecule to the specified FF, which is sometimes needed for some simulation engines to ensure the initial configuration energy is not too high.

Note: The energy minimize step requires the foyer[42] package.

```
hexane = mb.load('CCCCCC', smiles=True)
hexane.name = 'HEX'
hexane.energy_minimize(forcefield='oplsaa', steps=10**4)


ethanol = mb.load('CCO', smiles=True)
```

(continues on next page)

---

[38] https://foyer.mosdef.org/en/stable/
[39] https://mosdef.org
[40] https://foyer.mosdef.org/en/stable/
[41] https://www.daylight.com/dayhtml/doc/theory/theory.smiles.html
[42] https://foyer.mosdef.org/en/stable/

```
ethanol.name = 'ETO'
ethanol.energy_minimize(forcefield='oplsaa', steps=10**4)
```

The liquid box is built to a density of 680 kg/m³, with a 50/50 mol ratio of hexane and ethanol, and will be in an orthogonal box measuring 5.0 nm in the x, y, and z-dimensions.

```
box_liq = mb.fill_box(compound= [hexane, ethanol],
                      density=680,
                      compound_ratio=[0.5, 0.5],
                      box=[5.0, 5.0, 5.0])
```

United Atom (UA) Methane System

Note: foyer[43] is used in conjunction with mBuild in the following example to demonstrate how the MoSDeF[44] libraries integrate to generate a simulation box. A subset of the TraPPE-United Atom[45] force field (FF) comes standard with the foyer[46] software package.

Import the required mbuild package.

```
import mbuild as mb
```

Construct a pseudo-monatomic molecule (united atom (UA) methane), for use with the TraPPE[47] FF. The UA methane, bead type "_CH4", will be built as a child (mbuild.Compound.children), so the parent (mbuild.Compound) will allow a user-selected residue name (mbuild.Compound.name). If the methane is built using methane = mb.Compound(name="_CH4"), then the user must keep the residue name "_CH4" or foyer[48] will not recognize the bead type when using the standard TraPPE force field XML file.

```
methane = mb.Compound(name="MET")
methane_child_bead = mb.Compound(name="_CH4")
methane.add(methane_child_bead, inherit_periodicity=False)
```

Note: The inherit_periodicity flag is an optional boolean (default=True), which replaces the periodicity of self with the periodicity of the Compound being added.

The orthogonal liquid box contains 1230 methane molecules and measures 4.5 nm in all the x, y, and z-dimensions.

```
box_liq = mb.fill_box(compound=methane,
                      n_compounds=1230,
                      box=[4.5, 4.5, 4.5]
                      )
```

[43] https://foyer.mosdef.org/en/stable/
[44] https://mosdef.org
[45] http://trappe.oit.umn.edu
[46] https://foyer.mosdef.org/en/stable/
[47] http://trappe.oit.umn.edu
[48] https://foyer.mosdef.org/en/stable/

Polymer

Use two (2) different monomer units, A and B, to construct a polymer, capping it with a carboxylic acid and amine end group.

Import the required mbuild packages.

```
import mbuild as mb
from mbuild.lib.recipes.polymer import Polymer
```

Create the monomer units *comp_1* and *comp_2* using SMILES strings[49]. Set the *chain* as a `Polymer` class, adding *comp_1* and *comp_2* as the monomers A and B to the polymer.

Note: Setting the indices identifies which atoms will be removed and have ports created in their place.

```
comp_1 = mb.load('CC', smiles=True) # mBuild compound of the monomer unit
comp_2 = mb.load('COC', smiles=True) # mBuild compound of the monomer unit
chain = Polymer()
chain.add_monomer(compound=comp_1,
                  indices=[2, -2],
                  separation=.15,
                  replace=True)

chain.add_monomer(compound=comp_2,
                  indices=[3, -1],
                  separation=.15,
                  replace=True)
```

Select the carboxylic acid and amine end groups that we want to use for the head and tail of the polymer. Then, build the polymer with three (3) iterations of the AB sequence, and the selected head and tail end groups.

```
chain.add_end_groups(mb.load('C(=O)O',smiles=True),
                     index=3,
                     separation=0.15,
                     duplicate=False,
                     label="head")

chain.add_end_groups(mb.load('N', smiles=True),
                     index=-1,
                     separation=0.13,
                     duplicate=False,
                     label="tail")

chain.build(n=3, sequence='AB')
chain.visualize()
```

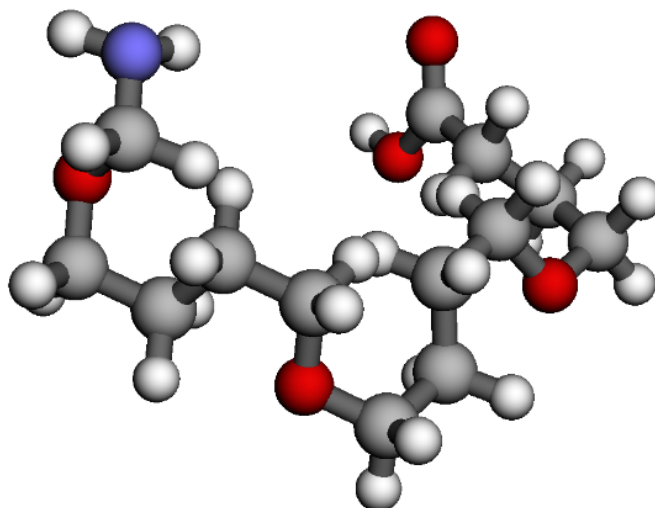[49] https://www.daylight.com/dayhtml/doc/theory/theory.smiles.html

Fig. 2: This **example polymer** is 3 of the AB sequences together with carboxylic acid and amine end groups.

1.4 File Writers

The mBuild library also supports simulation engine-specific file writers. These writers create a complete set of simulation writers to input files or a partial set of file writers, where the other required files are generated via another means.

mBuild utilizes ParmEd to write `Compound` information to a variety of file formats (e.g. PDB, MOL2, GRO. The full list of formats supported by ParmEd can be found at the ParmEd website[50]). Additionally, mBuild features several internal writers for file formats not yet supported by ParmEd. Information on these internal writers can be found below.

By default, many mBuild functions will only write coordinate and bond information to these files, i.e. no angles or dihedrals, and no atom typing is performed (atom names are used as atom types). However, force fields can be applied to Compounds by passing force field XML files (used by the Foyer package[51]) to the `Compound.save` function if Foyer is installed. If a force field is applied to a Compound, the mBuild internal writers will also write angle and dihedral information to the file in addition to labelling atoms by the atom types specified by the force field. The CHARMM-style GOMC writers (supported through the MoSDeF-GOMC extension[52]) are the exception to this default rule since they need a force field to build the files, as these files depend on the force field parameters (Example: charge and MW in the PSF files).

The simulation engine writers that use mBuild or are currently contained in the mBuild library:

- Cassandra[53]

- GROMACS[54]

- HOOMD-blue[55]

- Large-scale Atomic/Molecular Massively Parallel Simulator (LAMMPS)[56]

Support for GPU Optimized Monte Carlo (GOMC)[57] is also available through the MoSDeF-GOMC

---

[50] http://parmed.github.io/ParmEd/html/readwrite.html
[51] https://github.com/mosdef-hub/foyer
[52] https://github.com/GOMC-WSU/MoSDeF-GOMC
[53] https://cassandra.nd.edu/
[54] https://www.gromacs.org/
[55] http://glotzerlab.engin.umich.edu/hoomd-blue//
[56] https://lammps.sandia.gov/
[57] http://gomc.eng.wayne.edu/

library[58]

Cassandra File Writers

Cassandra Molecular Connectivity format.

https://cassandra-mc.readthedocs.io/en/latest/guides/input_files.html#molecular-connectivity-file

`mbuild.formats.cassandramcf.`**`write_mcf`**(*structure, filename, angle_style, dihedral_style, lj14=None, coul14=None*)

Output a Cassandra molecular connectivity file (MCF).

Outputs a Cassandra MCF from a Parmed structure object.

> **Parameters**
>
> > **structure**
> > [parmed.Structure] ParmEd structure object
> >
> > **filename**
> > [str] Path of the output file
> >
> > **angle_style**
> > [str] Type of angles. 'fixed' and 'harmonic' are valid choices
> >
> > **dihedral_style**
> > [str] Type of dihedrals. 'harmonic', 'OPLS', 'CHARMM', and 'none' are valid choices
> >
> > **lj14**
> > [float] Scaling factor for LJ interactions on 1-4 pairs
> >
> > **coul14**
> > [float] Scaling factor for Coulombic interactions on 1-4 pairs
>
> **Notes**
>
> See https://cassandra.nd.edu/index.php/documentation for a complete description of the MCF format.

HOOMD-blue File Writers

Write GSD (General Simulation Data)

Default data file format for HOOMD-blue

GSD format.

https://gsd.readthedocs.io/en/stable/

`mbuild.formats.gsdwriter.`**`write_gsd`**(*structure, filename, ref_distance=1.0, ref_mass=1.0, ref_energy=1.0, rigid_bodies=None, shift_coords=True, write_special_pairs=True, **kwargs*)

Output a GSD file (HOOMD v2 default data format).

> **Parameters**

---

[58] https://github.com/GOMC-WSU/MoSDeF-GOMC

structure
[parmed.Structure] ParmEd Structure object

filename
[str] Path of the output file.

ref_distance
[float, optional, default=1.0] Reference distance for conversion to reduced units

ref_mass
[float, optional, default=1.0] Reference mass for conversion to reduced units

ref_energy
[float, optional, default=1.0] Reference energy for conversion to reduced units

rigid_bodies
[list of int, optional, default=None] List of rigid body information. An integer value is required for each atom corresponding to the index of the rigid body the particle is to be associated with. A value of None indicates the atom is not part of a rigid body.

shift_coords
[bool, optional, default=True] Shift coordinates from (0, L) to (-L/2, L/2) if necessary.

write_special_pairs
[bool, optional, default=True] Writes out special pair information necessary to correctly use the OPLS fudged 1,4 interactions in HOOMD.

Notes

Force field parameters are not written to the GSD file and must be included manually into a HOOMD input script.

Create HOOMD-blue force field (>= 3.0)

HOOMD v3 forcefield format.

```
mbuild.formats.hoomd_forcefield.create_hoomd_forcefield(structure, r_cut,
                                                        ref_distance=1.0,
                                                        ref_mass=1.0,
                                                        ref_energy=1.0,
                                                        auto_scale=False,
                                                        nlist_buffer=0.4,
                                                        snapshot_kwargs={},
                                                        pppm_kwargs={'Nx':
                                                        8, 'Ny': 8, 'Nz': 8,
                                                        'order': 4},
                                                        init_snap=None)
```

Convert a parametrized pmd.Structure to a HOOMD snapshot and forces.

Parameters

structure
[parmed.Structure] ParmEd Structure object

r_cut
   [float] Cutoff radius in simulation units

ref_distance
   [float, optional, default=1.0] Reference distance for unit conversion ((Angstrom) / (desired units))

ref_mass
   [float, optional, default=1.0] Reference mass for unit conversion ((Dalton) / (desired units))

ref_energy
   [float, optional, default=1.0] Reference energy for unit conversion ((kcal/mol) / (desired units))

auto_scale
   [bool, optional, default=False] Scale to reduced units by automatically using the largest sigma value as ref_distance, largest mass value as ref_mass, and largest epsilon value as ref_energy

nlist_buffer
   [float, optional, default=True] buffer argument to pass to hoomd.md.nlist.Cell

snapshot_kwargs
   [dict] Keyword arguments to pass to to_hoomdsnapshot

pppm_kwargs
   [dict] Keyword arguments to pass to hoomd.md.long_range.pppm.make_pppm_coulomb_forces

init_snap
   [hoomd.Snapshot, optional, default=None] Initial snapshot to which to add the ParmEd structure object (useful for rigid bodies)

Returns

hoomd_snapshot
   [hoomd.Snapshot] HOOMD snapshot object to initialize the simulation

hoomd_forcefield
   [list[hoomd.md.force.Force]] List of hoomd force computes created during conversion

ReferenceValues
   [namedtuple] Values used in scaling

Notes

If you pass a non-parametrized pmd.Structure, you will not have angle, dihedral, or force field information. You may be better off creating a hoomd.Snapshot. See mbuild.formats.hoomd_snapshot.to_hoomdsnapshot()

About units: This method operates on a Parmed.Structure object where the units used differ from those used in mBuild and Foyer which may have been used when creating the typed Parmed.Structure.

The default units used when writing out the HOOMD Snapshot are: Distance (Angstrom) Mass (Dalton) Energy (kcal/mol)

If you wish to convert this unit system to another, you can use the reference parameters (ref_distance, ref_mass, ref_energy). The values used here should be ex-

pected to convert from the Parmed Structure units (above) to your desired units. The Parmed.Structure values are divided by the reference values.

If you wish to used a reduced unit system, set auto_scale = True. When auto_scale is True, the reference parameters won't be used.

Examples

To convert the energy units from kcal/mol to kj/mol:
 use ref_energy = 0.2390057 (kcal/kj)

To convert the distance units from Angstrom to nm:
 use ref_distance = 10 (angstroms/nm)

To use a reduced unit system, where mass, sigma, and epsilon are scaled by the largest value of each:

 use auto_scale = True, ref_distance = ref_energy = ref_mass = 1

Create HOOMD-blue Simulation (v2.x)

HOOMD simulation format.

```
mbuild.formats.hoomd_simulation.create_hoomd_simulation(structure, r_cut,
                                                        ref_distance=1.0,
                                                        ref_mass=1.0,
                                                        ref_energy=1.0,
                                                        auto_scale=False,
                                                        snapshot_kwargs={},
                                                        pppm_kwargs={'Nx':
                                                        8, 'Ny': 8, 'Nz': 8,
                                                        'order': 4},
                                                        init_snap=None,
                                                        restart=None,
                                                        nlist=<Mock
                                                        name='mock.md.nlist.cell'
                                                        id='140242667774416'>)
```

Convert a parametrized pmd.Structure to hoomd.SimulationContext.

 Parameters

  structure
   [parmed.Structure] ParmEd Structure object

  r_cut
   [float] Cutoff radius in simulation units

  ref_distance
   [float, optional, default=1.0] Reference distance for unit conversion (from Angstrom)

  ref_mass
   [float, optional, default=1.0] Reference mass for unit conversion (from Dalton)

  ref_energy
   [float, optional, default=1.0] Reference energy for unit conversion (from kcal/mol)

auto_scale

[bool, optional, default=False] Scale to reduced units by automatically using the largest sigma value as ref_distance, largest mass value as ref_mass, and largest epsilon value as ref_energy

snapshot_kwargs

[dict] Kwargs to pass to to_hoomdsnapshot

pppm_kwargs

[dict] Kwargs to pass to hoomd's pppm function

init_snap

[hoomd.data.SnapshotParticleData, optional, default=None] Initial snapshot to which to add the ParmEd structure object (useful for rigid bodies)

restart

[str, optional, default=None] Path to the gsd file from which to restart the simulation. https://hoomd-blue.readthedocs.io/en/v2.9.4/restartable-jobs.html Note: It is assumed that the ParmEd structure and the system in restart.gsd contain the same types. The ParmEd structure is still used to initialize the forces, but restart.gsd is used to initialize the system state (e.g., particle positions, momenta, etc).

nlist

[hoomd.md.nlist, default=hoomd.md.nlist.cell] Type of neighborlist to use, see https://hoomd-blue.readthedocs.io/en/stable/module-md-nlist.html for more information.

Returns

hoomd_objects

[list] List of hoomd objects created during conversion

ReferenceValues

[namedtuple] Values used in scaling

HOOMD-blue Snapshot

HOOMD snapshot format.

mbuild.formats.hoomd_snapshot.**from_snapshot**(*snapshot*, *scale=1.0*)

Convert a Snapshot to a Compound.

Snapshot can be a hoomd.Snapshot or a gsd.hoomd.Frame.

Parameters

snapshot

[hoomd.Snapshot or gsd.hoomd.Frame] Snapshot from which to build the mbuild Compound.

scale

[float, optional, default 1.0] Value by which to scale the length values

Returns

comp

[Compound]

`mbuild.formats.hoomd_snapshot.`**`to_hoomdsnapshot`**(*structure, ref_distance=1.0,*
ref_mass=1.0, ref_energy=1.0,
rigid_bodies=None,
shift_coords=True,
write_special_pairs=True,
auto_scale=False,
parmed_kwargs={},
hoomd_snapshot=None)

Convert a Compound or parmed.Structure to hoomd.Snapshot.

Parameters

structure
  [parmed.Structure] ParmEd Structure object Reference distance for unit conversion ((Angstrom) / (desired units))

ref_mass
  [float, optional, default=1.0] Reference mass for unit conversion ((Dalton) / (desired units))

ref_energy
  [float, optional, default=1.0] Reference energy for unit conversion ((kcal/mol) / (desired units))

rigid_bodies
  [list of int, optional, default=None] List of rigid body information. An integer value is required for each atom corresponding to the index of the rigid body the particle is to be associated with. A value of None indicates the atom is not part of a rigid body.

shift_coords
  [bool, optional, default=True] Shift coordinates from (0, L) to (-L/2, L/2) if necessary.

auto_scale
  [bool, optional, default=False] Automatically use largest sigma value as ref_distance, largest mass value as ref_mass and largest epsilon value as ref_energy

write_special_pairs
  [bool, optional, default=True] Writes out special pair information necessary to correctly use the OPLS fudged 1,4 interactions in HOOMD.

hoomd_snapshot
  [hoomd.Snapshot, optional, default=None] Initial snapshot to which to add the ParmEd structure object. The box information of the initial snapshot will be overwritten. (useful for rigid bodies)

Returns

hoomd_snapshot
  [hoomd.Snapshot]

ReferenceValues
  [namedtuple] Values used in scaling

Notes

This method does not create hoomd forcefield objects and the snapshot returned does not store the forcefield parameters. See mbuild.formats.hoomd_forcefield.create_hoomd_forcefield()

About units: This method operates on a Parmed.Structure object
> where the units used differ from those used in mBuild and Foyer which may have been used when creating the typed Parmed.Structure.

The default units used when writing out the HOOMD Snapshot are: Distance (Angstrom) Mass (Dalton) Energy (kcal/mol)

If you wish to convert this unit system to another, you can use the reference parameters (ref_distance, ref_mass, ref_energy). The values used here should be expected to convert from the Parmed Structure units (above) to your desired units. The Parmed.Structure values are divided by the reference values.

If you wish to used a reduced unit system, set auto_scale = True. When auto_scale is True, the reference parameters won't be used.

Examples

To convert the energy units from kcal/mol to kj/mol:
> use ref_energy = 0.2390057 (kcal/kj)

To convert the distance units from Angstrom to nm:
> use ref_distance = 10 (angstroms/nm)

To use a reduced unit system, where mass, sigma, and epsilon are scaled by the largest value of each:
> use auto_scale = True, ref_distance = ref_energy = ref_mass = 1

LAMMPS File Writers

Write Lammps data

LAMMPS data format.

`mbuild.formats.lammpsdata.`**`write_lammpsdata`**`(`*structure, filename, atom_style='full', unit_style='real', mins=None, maxs=None, pair_coeff_label=None, detect_forcefield_style=True, nbfix_in_data_file=True, use_urey_bradleys=False, use_rb_torsions=True, use_dihedrals=False, sigma_conversion_factor=None, epsilon_conversion_factor=None, mass_conversion_factor=None, charge_conversion_factor=True, zero_dihedral_weighting_factor=False, moleculeID_offset=1*`)`

Output a LAMMPS data file.

Outputs a LAMMPS data file in the 'full' atom style format. Default units are 'real' units. See http://lammps.sandia.gov/doc/atom_style.html for more information on atom styles.

Parameters

    structure
        [parmed.Structure] ParmEd structure object

    filename
        [str] Path of the output file

    atom_style: str, optional, default='full'
        Defines the style of atoms to be saved in a LAMMPS data file. The following atom styles are currently supported: 'full', 'atomic', 'charge', 'molecular' see http://lammps.sandia.gov/doc/atom_style. html for more information on atom styles.

    unit_style: str, optional, default='real'
        Defines to unit style to be save in a LAMMPS data file. Defaults to 'real' units. Current styles are supported: 'real', 'lj', 'metal' see https:// lammps.sandia.gov/doc/99/units.html for more information on unit styles

    mins
        [list, optional, default=None] Minimum box dimension in x, y, z directions, nm

    maxs
        [list, optional, default=None] Maximum box dimension in x, y, z directions, nm

    pair_coeff_label
        [str, optional, default=None] Provide a custom label to the pair_coeffs section in the lammps data file. A value of None means a suitable default will be chosen.

    detect_forcefield_style
        [bool, optional, default=True] If True, format lammpsdata parameters based on the contents of the parmed Structure

    use_urey_bradleys
        [bool, optional, default=False] If True, will treat angles as CHARMM-style angles with urey bradley terms while looking for *structure.urey_bradleys*

    use_rb_torsions
        [bool, optional, default=True] If True, will treat dihedrals OPLS-style torsions while looking for *structure.rb_torsions*

    use_dihedrals
        [bool, optional, default=False] If True, will treat dihedrals as CHARMM-style dihedrals while looking for *structure.dihedrals*

    zero_dihedral_weighting_factor
        [bool, optional, default=False] If True, will set weighting parameter to zero in CHARMM-style dihedrals. This should be True if the CHARMM dihedral style is used in non-CHARMM forcefields.

    sigma_conversion_factor
        [float, optional, default=None] If unit_style is set to 'lj', then sigma conversion factor is used to non-dimensionalize. Assume to be in

units of nm. If None, will take the largest sigma value in the structure.atoms.sigma values.

epsilon_conversion_factor
[float, optional, default=None] If unit_style is set to 'lj', then epsilon conversion factor is used to non-dimensionalize. Assume to be in units of kCal/mol. If None, will take the largest epsilon value in the structure.atoms.epsilon values.

mass_conversion_factor
[float, optional, default=None] If unit_style is set to 'lj', then mass conversion factor is used to non-dimensionalize. Assume to be in units of amu. If None, will take the largest mass value in the structure.atoms.mass values.

charge_conversion_factor
[bool, optional, default=True] If unit_style is set to 'lj', then charge conversion factor may or may not be used to non-dimensionalize. Assume to be in elementary charge units. If `True`, the charges are scaled by `np.sqrt(4*np.pi()*eps_0*sigma_conversion_factor*epsilon_conversion_factor)`. If `False`, the charges are not scaled and the user must be wary to choose the dielectric constant properly, which may be more convenient to implement an implicit solvent.

moleculeID_offset
[int , optional, default=1] Since LAMMPS treats the MoleculeID as an additional set of information to identify what molecule an atom belongs to, this currently behaves as a residue id. This value needs to start at 1 to be considered a real molecule.

Notes

See http://lammps.sandia.gov/doc/2001/data_format.html for a full description of the LAMMPS data format. Currently the following sections are supported (in addition to the header): *Masses, Nonbond Coeffs, Bond Coeffs, Angle Coeffs, Dihedral Coeffs, Atoms, Bonds, Angles, Dihedrals, Impropers* OPLS and CHARMM forcefield styles are supported, AMBER forcefield styles are NOT

Some of this function has beed adopted from *mdtraj*'s support of the LAMMPSTRJ trajectory format. See https://github.com/mdtraj/mdtraj/blob/master/mdtraj/formats/lammpstrj.py for details.

unique_types
[a sorted list of unique atomtypes for all atoms in the structure.]

Defined by:
atomtype : atom.type

unique_bond_types: an enumarated OrderedDict of unique bond types for all bonds in the structure.
Defined by bond parameters and component atomtypes, in order: — k : bond.type.k — req : bond.type.req — atomtypes : sorted((bond.atom1.type, bond.atom2.type))

unique_angle_types: an enumerated OrderedDict of unique angle types for all
angles in the structure. Defined by angle parameters and component atomtypes, in order: — k : angle.type.k — theteq : angle.type.theteq — vertex atomtype: angle.atom2.type — atomtypes: sorted((bond.atom1.type, bond.atom3.type))

unique_dihedral_types: an enumerated OrderedDict of unique dihedrals type
for all dihedrals in the structure. Defined by dihedral parameters and component atomtypes, in order: — c0 : dihedral.type.c0 — c1 : dihedral.type.c1 — c2 : dihedral.type.c2 — c3 : dihedral.type.c3 — c4 : dihedral.type.c4 — c5 : dihedral.type.c5 — scee : dihedral.type.scee — scnb : dihedral.type.scnb — atomtype 1 : dihedral.atom1.type — atomtype 2 : dihedral.atom2.type — atomtype 3 : dihedral.atom3.type — atomtype 4 : dihedral.atom4.type

1.5 Tutorials

Methane: Compounds and bonds

Note: mBuild expects all distance units to be in nanometers.

The primary building block in mBuild is a `Compound`. Anything you construct will inherit from this class. Let's start with some basic imports and initialization:

```python
import mbuild as mb

class Methane(mb.Compound):
    def __init__(self):
        super(Methane, self).__init__()
```

Any `Compound` can contain other `Compounds` which can be added using its `add()` method. `Compounds` at the bottom of such a hierarchy are referred to as `Particles`. Note however, that this is purely semantic in mBuild to help clearly designate the bottom of a hierarchy.

```python
import mbuild as mb

class Methane(mb.Compound):
    def __init__(self):
        super(Methane, self).__init__()
        carbon = mb.Particle(name='C')
        self.add(carbon, label='C[$]')

        hydrogen = mb.Particle(name='H', pos=[0.11, 0, 0])
        self.add(hydrogen, label='HC[$]')
```

By default a created `Compound`/`Particle` will be placed at `0, 0, 0` as indicated by its `pos` attribute. The `Particle` objects contained in a `Compound`, the bottoms of the hierarchy, can be referenced via the `particles` method which returns a generator of all `Particle` objects contained below the `Compound` in the hierarchy.

Note: All positions in mBuild are stored in nanometers.

Any part added to a `Compound` can be given an optional, descriptive string label. If the label ends with the characters `[$]`, a list will be created in the labels. Any subsequent parts added to the `Compound` with the same label prefix will be appended to the list. In the example above, we've labeled the hydrogen as `HC[$]`. So this first part, with the label prefix `HC`, is now referenceable via `self['HC'][0]`. The next part added with the label `HC[$]` will be referenceable via `self['HC'][1]`.

Now let's use these styles of referencing to connect the carbon to the hydrogen. Note that for typical use cases, you will almost never have to explicitly define a bond when using mBuild - this is just to show you what's going on under the hood:

```python
import mbuild as mb

class Methane(mb.Compound):
    def __init__(self):
        super(Methane, self).__init__()
        carbon = mb.Particle(name='C')
        self.add(carbon, label='C[$]')

        hydrogen = mb.Particle(name='H', pos=[0.11, 0, 0])
        self.add(hydrogen, label='HC[$]')

        self.add_bond((self[0], self['HC'][0]))
```

As you can see, the carbon is placed in the zero index of self. The hydrogen could be referenced via self[1] but since we gave it a fancy label, it's also referenceable via self['HC'][0].

Alright now that we've got the basics, let's finish building our Methane and take a look at it:

```python
import mbuild as mb

class Methane(mb.Compound):
    def __init__(self):
        super(Methane, self).__init__()
        carbon = mb.Particle(name='C')
        self.add(carbon, label='C[$]')

        hydrogen = mb.Particle(name='H', pos=[0.1, 0, -0.07])
        self.add(hydrogen, label='HC[$]')

        self.add_bond((self[0], self['HC'][0]))

        self.add(mb.Particle(name='H', pos=[-0.1, 0, -0.07]), label='HC[$]')
        self.add(mb.Particle(name='H', pos=[0, 0.1, 0.07]), label='HC[$]')
        self.add(mb.Particle(name='H', pos=[0, -0.1, 0.07]), label='HC[$]')

        self.add_bond((self[0], self['HC'][1]))
        self.add_bond((self[0], self['HC'][2]))
        self.add_bond((self[0], self['HC'][3]))
```

```python
methane = Methane()
methane.visualize()
```

```python
# Save to .mol2
methane.save('methane.mol2',overwrite=True)
```

Ethane: Reading from files, Ports and coordinate transforms

Note: mBuild expects all distance units to be in nanometers.

In this example, we'll cover reading molecular components from files, introduce the concept of `Ports` and start using some coordinate transforms.

First, we need to import the mbuild package:

```
import mbuild as mb
```

As you probably noticed while creating your methane molecule in the last tutorial, manually adding `Particles` and `Bonds` to a `Compound` is a bit cumbersome. The easiest way to create small, reusable components, such as methyls, amines or monomers, is to hand draw them using software like Avogadro[59] and export them as either a .pdb or .mol2 file (the file should contain connectivity information).

Let's start by reading a methyl group from a `.pdb` file:

```
import mbuild as mb

ch3 = mb.load('ch3.pdb')
ch3.visualize()
```

Now let's use our first coordinate transform to center the methyl at its carbon atom:

```
import mbuild as mb

ch3 = mb.load('ch3.pdb')
ch3.translate(-ch3[0].pos)  # Move carbon to origin.
```

Now we have a methyl group loaded up and centered. In order to connect `Compounds` in mBuild, we make use of a special type of `Compound`: the `Port`. A `Port` is a `Compound` with two sets of four "ghost" `Particles` that assist in bond creation. In addition, `Ports` have an `anchor` attribute which typically points to a particle that the `Port` should be associated with. In our methyl group, the `Port` should be anchored to the carbon atom so that we can now form bonds to this carbon:

```
import mbuild as mb

ch3 = mb.load('ch3.pdb')
ch3.translate(-ch3[0].pos)  # Move carbon to origin.

port = mb.Port(anchor=ch3[0])
ch3.add(port, label='up')

# Place the port at approximately half a C-C bond length.
ch3['up'].translate([0, -0.07, 0])
```

By default, `Ports` are never output from the mBuild structure. However, it can be useful to look at a molecule with the `Ports` to check your work as you go:

```
ch3.visualize(show_ports=True)
```

Now we wrap the methyl group into a python class, so that we can reuse it as a component to build more complex molecules later.

---

[59] https://avogadro.cc/

```python
import mbuild as mb


class CH3(mb.Compound):
    def __init__(self):
        super(CH3, self).__init__()

        mb.load('ch3.pdb', compound=self)
        self.translate(-self[0].pos)  # Move carbon to origin.

        port = mb.Port(anchor=self[0])
        self.add(port, label='up')
        # Place the port at approximately half a C-C bond length.
        self['up'].translate([0, -0.07, 0])
```

When two `Ports` are connected, they are forced to overlap in space and their parent `Compounds` are rotated and translated by the same amount.

Note: If we tried to connect two of our methyls right now using only one set of four ghost particles, not only would the `Ports` overlap perfectly, but the carbons and hydrogens would also perfectly overlap - the 4 ghost atoms in the `Port` are arranged identically with respect to the other atoms. For example, if a `Port` and its direction is indicated by "<-", forcing the port in <-CH3 to overlap with <-CH3 would just look like <-CH3 (perfectly overlapping atoms).

To solve this problem, every port contains a second set of 4 ghost atoms pointing in the opposite direction. When two `Compounds` are connected, the port that places the anchor atoms the farthest away from each other is chosen automatically to prevent this overlap scenario.

When <->CH3 and <->CH3 are forced to overlap, the CH3<->CH3 is automatically chosen.

Now the fun part: stick 'em together to create an ethane:

```python
ethane = mb.Compound()

ethane.add(CH3(), label="methyl_1")
ethane.add(CH3(), label="methyl_2")
mb.force_overlap(move_this=ethane['methyl_1'],
                 from_positions=ethane['methyl_1']['up'],
                 to_positions=ethane['methyl_2']['up'])
```

Above, the `force_overlap()` function takes a `Compound` and then rotates and translates it such that two other `Compounds` overlap. Typically, as in this case, those two other `Compounds` are `Ports` - in our case, `methyl1['up']` and `methyl2['up']`.

```python
ethane.visualize()
```

```python
ethane.visualize(show_ports=True)
```

Similarly, if we want to make ethane a reusable component, we need to wrap it into a python class.

```python
import mbuild as mb


class Ethane(mb.Compound):
    def __init__(self):
        super(Ethane, self).__init__()

        self.add(CH3(), label="methyl_1")
        self.add(CH3(), label="methyl_2")
```

```
            mb.force_overlap(move_this=self['methyl_1'],
                             from_positions=self['methyl_1']['up'],
                             to_positions=self['methyl_2']['up'])
```

```
ethane = Ethane()
ethane.visualize()
```

```
# Save to .mol2
ethane.save('ethane.mol2', overwrite=True)
```

Monolayer: Complex hierarchies, patterns, tiling and writing to files

Note: mBuild expects all distance units to be in nanometers.

In this example, we'll cover assembling more complex hierarchies of components using patterns, tiling and how to output systems to files. To illustrate these concepts, let's build an alkane monolayer on a crystalline substrate.

First, let's build our monomers and functionalized them with a silane group which we can then attach to the substrate. The `Alkane` example uses the `polymer` tool to combine CH2 and CH3 repeat units. You also have the option to cap the front and back of the chain or to leave a CH2 group with a dangling port. The `Silane` compound is a Si(OH)₂ group with two ports facing out from the central Si. Lastly, we combine `alkane` with `silane` and add a label to `AlkylSilane` which points to, `silane['down']`. This allows us to reference it later using `AlkylSilane['down']` rather than `AlkylSilane['silane']['down']`.

Note: In `Compounds` with multiple `Ports`, by convention, we try to label every `Port` successively as 'up', 'down', 'left', 'right', 'front', 'back' which should roughly correspond to their relative orientations. This is a bit tricky to enforce because the system is so flexible so use your best judgement and try to be consistent! The more components we collect in our library with the same labeling conventions, the easier it becomes to build ever more complex structures.

```
import mbuild as mb

from mbuild.lib.recipes import Alkane
from mbuild.lib.moieties import Silane


class AlkylSilane(mb.Compound):
    """A silane functionalized alkane chain with one Port. """
    def __init__(self, chain_length):
        super(AlkylSilane, self).__init__()

        alkane = Alkane(chain_length, cap_end=False)
        self.add(alkane, 'alkane')
        silane = Silane()
        self.add(silane, 'silane')
        mb.force_overlap(self['alkane'], self['alkane']['down'], self['silane'][
→'up'])

        # Hoist silane port to AlkylSilane level.
        self.add(silane['down'], 'down', containment=False)
```

27

```
AlkylSilane(5).visualize()
```

Now let's create a substrate to which we can later attach our monomers:

```python
import mbuild as mb
from mbuild.lib.surfaces import Betacristobalite

surface = Betacristobalite()
tiled_surface = mb.lib.recipes.TiledCompound(surface, n_tiles=(2, 1, 1))
```

Here we've imported a beta-cristobalite surface from our component library. The `TiledCompound` tool allows you replicate any `Compound` in the x-, y- and z-directions by any number of times - 2, 1 and 1 for our case.

Next, let's create our monomer and a hydrogen atom that we'll place on unoccupied surface sites:

```python
from mbuild.lib.atoms import H
alkylsilane = AlkylSilane(chain_length=10)
hydrogen = H()
```

Then we need to tell mBuild how to arrange the chains on the surface. This is accomplished with the "pattern" tools. Every pattern is just a collection of points. There are all kinds of patterns like spherical, 2D, regular, irregular etc. When you use the `apply_pattern` command, you effectively superimpose the pattern onto the host compound, mBuild figures out what the closest ports are to the pattern points and then attaches copies of the guest onto the binding sites identified by the pattern:

```python
pattern = mb.Grid2DPattern(8, 8)  # Evenly spaced, 2D grid of points.

# Attach chains to specified binding sites. Other sites get a hydrogen.
chains, hydrogens = pattern.apply_to_compound(host=tiled_surface,
 →guest=alkylsilane, backfill=hydrogen)
```

Also note the `backfill` optional argument which allows you to place a different compound on any unused ports. In this case we want to backfill with hydrogen atoms on every port without a chain.

```python
monolayer = mb.Compound([tiled_surface, chains, hydrogens])
monolayer.visualize() # Warning: may be slow in IPython notebooks
```

```python
# Save as .mol2 file
monolayer.save('monolayer.mol2', overwrite=True)
```

`lib.recipes.monolayer.py` wraps many these functions into a simple, general class for generating the monolayers, as shown below:

```python
from mbuild.lib.recipes import Monolayer

monolayer = Monolayer(fractions=[1.0], chains=alkylsilane, backfill=hydrogen,
                      pattern=mb.Grid2DPattern(n=8, m=8),
                      surface=surface, tile_x=2, tile_y=1)
monolayer.visualize()
```

Point Particles: Basic system initialization

Note: mBuild expects all distance units to be in nanometers.

This tutorial focuses on the usage of basic system initialization operations, as applied to simple point particle systems (i.e., generic Lennard-Jones particles rather than specific atoms).

The code below defines several point particles in a cubic arrangement. Note, the color and radius associated with a Particle name can be set and passed to the visualize command. Colors are passed in hex format (see http://www.color-hex.com/color/bfbfbf).

```python
import mbuild as mb

class MonoLJ(mb.Compound):
    def __init__(self):
        super(MonoLJ, self).__init__()
        lj_particle1 = mb.Particle(name='LJ', pos=[0, 0, 0])
        self.add(lj_particle1)

        lj_particle2 = mb.Particle(name='LJ', pos=[1, 0, 0])
        self.add(lj_particle2)

        lj_particle3 = mb.Particle(name='LJ', pos=[0, 1, 0])
        self.add(lj_particle3)

        lj_particle4 = mb.Particle(name='LJ', pos=[0, 0, 1])
        self.add(lj_particle4)

        lj_particle5 = mb.Particle(name='LJ', pos=[1, 0, 1])
        self.add(lj_particle5)

        lj_particle6 = mb.Particle(name='LJ', pos=[1, 1, 0])
        self.add(lj_particle6)

        lj_particle7 = mb.Particle(name='LJ', pos=[0, 1, 1])
        self.add(lj_particle7)

        lj_particle8 = mb.Particle(name='LJ', pos=[1, 1, 1])
        self.add(lj_particle8)

monoLJ = MonoLJ()
monoLJ.visualize()
```

While this would work for defining a single molecule or very small system, this would not be efficient for large systems. Instead, the clone and translate operator can be used to facilitate automation. Below, we simply define a single prototype particle (lj_proto), which we then copy and translate about the system.

Note, mBuild provides two different translate operations, "translate" and "translate_to". "translate" moves a particle by adding the vector the original position, whereas "translate_to" move a particle to the specified location in space. Note, "translate_to" maintains the internal spatial relationships of a collection of particles by first shifting the center of mass of the collection of particles to the origin, then translating to the specified location. Since the lj_proto particle in this example starts at the origin, these two commands produce identical behavior.

```python
import mbuild as mb

class MonoLJ(mb.Compound):
    def __init__(self):
        super(MonoLJ, self).__init__()
        lj_proto = mb.Particle(name='LJ', pos=[0, 0, 0])

        for i in range(0,2):
            for j in range(0,2):
                for k in range(0,2):
                    lj_particle = mb.clone(lj_proto)
                    pos = [i,j,k]
                    lj_particle.translate(pos)
                    self.add(lj_particle)

monoLJ = MonoLJ()
monoLJ.visualize()
```

To simplify this process, mBuild provides several build-in patterning tools, where for example, Grid3DPattern can be used to perform this same operation. Grid3DPattern generates a set of points, from 0 to 1, which get stored in the variable "pattern". We need only loop over the points in pattern, cloning, translating, and adding to the system. Note, because Grid3DPattern defines points between 0 and 1, they must be scaled based on the desired system size, i.e., pattern.scale(2).

```python
import mbuild as mb

class MonoLJ(mb.Compound):
    def __init__(self):
        super(MonoLJ, self).__init__()
        lj_proto = mb.Particle(name='LJ', pos=[0, 0, 0])

        pattern = mb.Grid3DPattern(2, 2, 2)
        pattern.scale(2)

        for pos in pattern:
            lj_particle = mb.clone(lj_proto)
            lj_particle.translate(pos)
            self.add(lj_particle)

monoLJ = MonoLJ()
monoLJ.visualize()
```

Larger systems can therefore be easily generated by toggling the values given to Grid3DPattern. Other patterns can also be generated using the same basic code, such as a 2D grid pattern:

```python
import mbuild as mb

class MonoLJ(mb.Compound):
    def __init__(self):
        super(MonoLJ, self).__init__()
        lj_proto = mb.Particle(name='LJ', pos=[0, 0, 0])

        pattern = mb.Grid2DPattern(5, 5)
        pattern.scale(5)
```

```python
        for pos in pattern:
            lj_particle = mb.clone(lj_proto)
            lj_particle.translate(pos)
            self.add(lj_particle)

monoLJ = MonoLJ()
monoLJ.visualize()
```

Points on a sphere can be generated using SpherePattern. Points on a disk using DisKPattern, etc.

Note to show both simultaneously, we shift the x-coordinate of Particles in the sphere by -1 (i.e., pos[0]-=1.0) and +1 for the disk (i.e, pos[0]+=1.0).

```python
import mbuild as mb

class MonoLJ(mb.Compound):
    def __init__(self):
        super(MonoLJ, self).__init__()
        lj_proto = mb.Particle(name='LJ', pos=[0, 0, 0])

        pattern_sphere = mb.SpherePattern(200)
        pattern_sphere.scale(0.5)

        for pos in pattern_sphere:
            lj_particle = mb.clone(lj_proto)
            pos[0]-=1.0
            lj_particle.translate(pos)
            self.add(lj_particle)

        pattern_disk = mb.DiskPattern(200)
        pattern_disk.scale(0.5)
        for pos in pattern_disk:
            lj_particle = mb.clone(lj_proto)
            pos[0]+=1.0
            lj_particle.translate(pos)
            self.add(lj_particle)

monoLJ = MonoLJ()
monoLJ.visualize()
```

We can also take advantage of the hierachical nature of mBuild to accomplish the same task more cleanly. Below we create a component that corresponds to the sphere (class SphereLJ), and one that corresponds to the disk (class DiskLJ), and then instantiate and shift each of these individually in the MonoLJ component.

```python
import mbuild as mb

class SphereLJ(mb.Compound):
    def __init__(self):
        super(SphereLJ, self).__init__()
        lj_proto = mb.Particle(name='LJ', pos=[0, 0, 0])

        pattern_sphere = mb.SpherePattern(200)
```

```python
        pattern_sphere.scale(0.5)

        for pos in pattern_sphere:
            lj_particle = mb.clone(lj_proto)
            lj_particle.translate(pos)
            self.add(lj_particle)

class DiskLJ(mb.Compound):
    def __init__(self):
        super(DiskLJ, self).__init__()
        lj_proto = mb.Particle(name='LJ', pos=[0, 0, 0])

        pattern_disk = mb.DiskPattern(200)
        pattern_disk.scale(0.5)
        for pos in pattern_disk:
            lj_particle = mb.clone(lj_proto)
            lj_particle.translate(pos)
            self.add(lj_particle)


class MonoLJ(mb.Compound):
    def __init__(self):
        super(MonoLJ, self).__init__()

        sphere = SphereLJ();
        pos=[-1, 0, 0]
        sphere.translate(pos)
        self.add(sphere)

        disk = DiskLJ();
        pos=[1, 0, 0]
        disk.translate(pos)
        self.add(disk)


monoLJ = MonoLJ()
monoLJ.visualize()
```

Again, since mBuild is hierarchical, the pattern functions can be used to generate large systems of any arbitary component. For example, we can replicate the SphereLJ component on a regular array.

```python
import mbuild as mb

class SphereLJ(mb.Compound):
    def __init__(self):
        super(SphereLJ, self).__init__()
        lj_proto = mb.Particle(name='LJ', pos=[0, 0, 0])

        pattern_sphere = mb.SpherePattern(13)
        pattern_sphere.scale(0.1)

        for pos in pattern_sphere:
            lj_particle = mb.clone(lj_proto)
```

```python
            lj_particle.translate(pos)
            self.add(lj_particle)
class MonoLJ(mb.Compound):
    def __init__(self):
        super(MonoLJ, self).__init__()
        sphere = SphereLJ();

        pattern = mb.Grid3DPattern(3, 3, 3)
        pattern.scale(2)

        for pos in pattern:
            lj_sphere = mb.clone(sphere)
            lj_sphere.translate_to(pos)
            #shift the particle so the center of mass
            #of the system is at the origin
            lj_sphere.translate([-5,-5,-5])

            self.add(lj_sphere)

monoLJ = MonoLJ()
monoLJ.visualize()
```

Several functions exist for rotating compounds. For example, the spin command allows a compound to be rotated, in place, about a specific axis (i.e., it considers the origin for the rotation to lie at the compound's center of mass).

```python
import mbuild as mb
import random
from numpy import pi


class CubeLJ(mb.Compound):
    def __init__(self):
        super(CubeLJ, self).__init__()
        lj_proto = mb.Particle(name='LJ', pos=[0, 0, 0])

        pattern = mb.Grid3DPattern(2, 2, 2)
        pattern.scale(0.2)

        for pos in pattern:
            lj_particle = mb.clone(lj_proto)
            lj_particle.translate(pos)
            self.add(lj_particle)

class MonoLJ(mb.Compound):
    def __init__(self):
        super(MonoLJ, self).__init__()
        cube_proto = CubeLJ();

        pattern = mb.Grid3DPattern(3, 3, 3)
        pattern.scale(2)
        rnd = random.Random()
        rnd.seed(123)
```

```python
        for pos in pattern:
            lj_cube = mb.clone(cube_proto)
            lj_cube.translate_to(pos)
            #shift the particle so the center of mass
            #of the system is at the origin
            lj_cube.translate([-5,-5,-5])
            lj_cube.spin( rnd.uniform(0, 2 * pi), [1, 0, 0])
            lj_cube.spin(rnd.uniform(0, 2 * pi), [0, 1, 0])
            lj_cube.spin(rnd.uniform(0, 2 * pi), [0, 0, 1])

            self.add(lj_cube)

monoLJ = MonoLJ()
monoLJ.visualize()
```

Configurations can be dumped to file using the save command; this takes advantage of MDTraj and supports a range of file formats (see http://MDTraj.org).

```python
#save as xyz file
monoLJ.save('output.xyz')
#save as mol2
monoLJ.save('output.mol2')
```

Building a Simple Alkane

The purpose of this tutorial is to demonstrate the construction of an alkane polymer and provide familiarity with many of the underlying functions in mBuild. Note that a robust polymer construction recipe already exists in mBuild, which will also be demonstrated at the end of the tutorial.

Setting up the monomer

The first step is to construct the basic repeat unit for the alkane, i.e., a CH_2 group, similar to the construction of the CH_3 monomer in the prior methane tutorial. Rather than importing the coordinates from a pdb file, as in the previous example, we will instead explicitly define them in the class. Recall that distance units are nm in mBuild.

```python
import mbuild as mb

class CH2(mb.Compound):
    def __init__(self):
        super(CH2, self).__init__()
        # Add carbon
        self.add(mb.Particle(name='C', pos=[0,0,0]), label='C[$]')

        # Add hydrogens
        self.add(mb.Particle(name='H', pos=[-0.109, 0, 0.0]), label='HC[$]')
        self.add(mb.Particle(name='H', pos=[0.109, 0, 0.0]), label='HC[$]')

        # Add bonds between the atoms
        self.add_bond((self['C'][0], self['HC'][0]))
        self.add_bond((self['C'][0], self['HC'][1]))
```

```python
        # Add ports anchored to the carbon
        self.add(mb.Port(anchor=self[0]), label='up')
        self.add(mb.Port(anchor=self[0]), label='down')

        # Move the ports approximately half a C-C bond length away from the
→carbon
        self['up'].translate([0, -0.154/2, 0])
        self['down'].translate([0, 0.154/2, 0])

monomer = CH2()
monomer.visualize(show_ports=True)
```

This configuration of the monomer is not a particularly realistic conformation. One could use this monomer to construct a polymer and then apply an energy minimization scheme, or, as we will demonstrate here, we can use mBuild's rotation commands to provide a more realistic starting point.

Below, we use the same basic script, but now apply a rotation to the hydrogen atoms. Since the hydrogens start 180° apart and we know they should be ~109.5° apart, each should be rotated half of the difference closer to each other around the y-axis. Note that the rotation angle is given in radians. Similarly, the ports should be rotated around the x-axis by the same amount so that atoms can be added in a realistic orientation.

```python
import numpy as np
import mbuild as mb

class CH2(mb.Compound):
    def __init__(self):
        super(CH2, self).__init__()
        # Add carbon
        self.add(mb.Particle(name='C', pos=[0,0,0]), label='C[$]')

        # Add hydrogens
        self.add(mb.Particle(name='H', pos=[-0.109, 0, 0.0]), label='HC[$]')
        self.add(mb.Particle(name='H', pos=[0.109, 0, 0.0]), label='HC[$]')

        # Rotate the hydrogens
        theta = 0.5 * (180 - 109.5) * np.pi / 180
        #mb.rotate(self['HC'][0], theta, around=[0, 1, 0])
        #mb.rotate(self['HC'][1], -theta, around=[0, 1, 0])
        self['HC'][0].rotate( theta, around=[0, 1, 0])
        self['HC'][1].rotate(-theta, around=[0, 1, 0])

        # Add bonds between the atoms
        self.add_bond((self['C'][0], self['HC'][0]))
        self.add_bond((self['C'][0], self['HC'][1]))

        # Add the ports and appropriately rotate them
        self.add(mb.Port(anchor=self[0]), label='up')
        self['up'].translate([0, -0.154/2, 0])
        self['up'].rotate(theta, around=[1, 0, 0])

        self.add(mb.Port(anchor=self[0]), label='down')
        self['down'].translate([0, 0.154/2, 0])
```

```
        self['down'].rotate(-theta, around=[1, 0, 0])

monomer = CH2()
monomer.visualize(show_ports=True)
```

Defining the polymerization class

With a basic monomer construct, we can now construct a polymer by connecting the ports together. Here, we first instantiate one instance of the CH2 class as `1ast_monomer`, then use the clone function to make a copy. The `force_overlap()` function is used to connect the 'up' port from `current_monomer` to the 'down' port of `last_mononer`.

```
class AlkanePolymer(mb.Compound):
    def __init__(self):
        super(AlkanePolymer, self).__init__()
        last_monomer = CH2()
        self.add(last_monomer)
        for i in range(3):
            current_monomer = CH2()
            mb.force_overlap(move_this=current_monomer,
                             from_positions=current_monomer['up'],
                             to_positions=last_monomer['down'])
            self.add(current_monomer)
            last_monomer = current_monomer

polymer = AlkanePolymer()
polymer.visualize(show_ports=True)
```

Visualization of this structure demonstrates a problem; the polymer curls up on itself. This is a result of the fact that ports not only define the location in space, but also an orientation. This can be trivially fixed, by rotating the down port 180° around the y-axis.

We can also add a variable `chain_length` both to the for loop and `init` that will allow the length of the polymer to be adjusted when the class is instantiated.

```
import numpy as np
import mbuild as mb

class CH2(mb.Compound):
    def __init__(self):
        super(CH2, self).__init__()
        # Add carbons and hydrogens
        self.add(mb.Particle(name='C', pos=[0,0,0]), label='C[$]')
        self.add(mb.Particle(name='H', pos=[-0.109, 0, 0.0]), label='HC[$]')
        self.add(mb.Particle(name='H', pos=[0.109, 0, 0.0]), label='HC[$]')

        # rotate hydrogens
        theta = 0.5 * (180 - 109.5) * np.pi / 180
        self['HC'][0].rotate(theta, around=[0, 1, 0])
        self['HC'][1].rotate(-theta, around=[0, 1, 0])

        # Add bonds between the atoms
        self.add_bond((self['C'][0], self['HC'][0]))
```

```python
        self.add_bond((self['C'][0], self['HC'][1]))

        # Add ports
        self.add(mb.Port(anchor=self[0]), label='up')
        self['up'].translate([0, -0.154/2, 0])
        self['up'].rotate(theta, around=[1, 0, 0])

        self.add(mb.Port(anchor=self[0]), label='down')
        self['down'].translate([0, 0.154/2, 0])
        self['down'].rotate(np.pi, [0, 1, 0])
        self['down'].rotate(-theta, around=[1, 0, 0])


class AlkanePolymer(mb.Compound):
    def __init__(self, chain_length=1):
        super(AlkanePolymer, self).__init__()
        last_monomer = CH2()
        self.add(last_monomer)
        for i in range (chain_length-1):
            current_monomer = CH2()

            mb.force_overlap(move_this=current_monomer,
                             from_positions=current_monomer['up'],
                             to_positions=last_monomer['down'])
            self.add(current_monomer)
            last_monomer=current_monomer
```

```python
polymer = AlkanePolymer(chain_length=10)
polymer.visualize(show_ports=True)
```

Using mBuild's Polymer Class

mBuild provides a prebuilt class to perform this basic functionality. Since it is designed to be more general, it takes as an argument not just the replicates (n), sequence ('A' for a single monomer or 'AB' for two different monomers). Then, it binds them together by removing atom/bead via specifying its index number (indices). A graphical description of the polymer builder creating ports, then bonding them together is provided below.

Note: The port locations may be critical to ensure the molecule is not overlapping when it is built.

Building a Simple Hexane

A simple hexane molecule is built using mBuild's packaged polymer builder. This is done by loading a methane molecule via a SMILES string. The indices are explicitly selected, so the molecule builds out in the proper directions and does not overlap.

```python
import mbuild as mb
from mbuild.lib.recipes.polymer import Polymer

comp = mb.load('C', smiles=True) # mBuild compound of the monomer unit
```

```python
import mbuild as mb
from mbuild.lib.recipes.polymer import Polymer

monomer = mb.load('CC', smiles=True)
end_group = mb.load('C(=O)O', smiles=True)
chain = Polymer()

chain.add_monomer(compound=monomer,
                  indices=[2, 7],
                  separation=0.15,
                  replace=True)

chain.add_end_groups(end_group,
                     index=3,
                     separation=0.15)

chain.build(n=3, sequence='A')
```
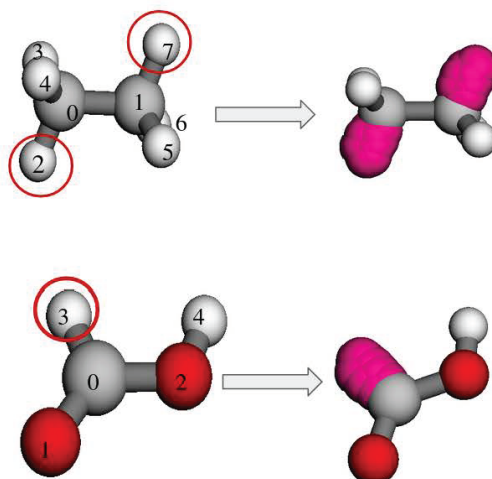
Fig. 3: **Polymer builder class example.** This shows how to define the atoms, which are replaced with ports. The ports are then bonded together between the monomers. Additionally, these ports can be utilized for adding different end groups moieties to the polymer.

```python
chain = Polymer()

chain.add_monomer(compound=comp,
                  indices=[1, -2],
                  separation=.15,
                  replace=True)

chain.build(n=6, sequence='A')
```

Using Multiple Monomers and Capping the Ends of a Polymer

This example uses methyl ether and methane monomers to build a polymer, capping it with fluorinated and alcohol end groups. The monomers are combined together in the 'AB' sequence two times (n=2), which means the polymer will contain 2 of each monomer (ABAB). The end groups are added via the add_end_groups attribute, specifying the atom to use (index), the distance of the bond (separation), the location of each end group (label), and if the tail end group is duplicated to the head of the polymer (duplicate). The indices are explicitly selected, so the molecule builds out in the proper directions and does not overlap.

```python
from mbuild.lib.recipes.polymer import Polymer
import mbuild as mb

comp_1 = mb.load('C', smiles=True)
comp_2 = mb.load('COC', smiles=True)
chain = Polymer()

chain.add_monomer(compound=comp_1,
                  indices=[1, -1],
                  separation=.15,
                  replace=True)
```

```
chain.add_monomer(compound=comp_2,
                  indices=[3, -1],
                  separation=.15,
                  replace=True)


chain.add_end_groups(mb.load('O',smiles=True), # Capping off this polymer with an␣
↪Alcohol
                     index=1,
                     separation=0.15, label="head", duplicate=False)

chain.add_end_groups(mb.load('F',smiles=True), # Capping off this polymer with a␣
↪Fluorine
                     index=1,
                     separation=0.18, label="tail", duplicate=False)


chain.build(n=2, sequence='AB')
chain.visualize(show_ports=True)
```

Building a System of Alkanes

A system of alkanes can be constructed by simply cloning the polymer constructed above and trans-
lating and/or rotating the alkanes in space. mBuild provides many routines that can be used to create
different patterns, to which the polymers can be shifted.

```
comp = mb.load('C', smiles=True) # mBuild compound of the monomer unit
polymer = Polymer()

polymer.add_monomer(compound=comp,
                    indices=[1, -2],
                    separation=.15,
                    replace=True)

polymer.build(n=10, sequence='A')

# the pattern we generate puts points in the xy-plane, so we'll rotate the␣
↪polymer
# so that it is oriented normal to the xy-plane
polymer.rotate(np.pi/2, [1, 0, 0])

# define a compound to hold all the polymers
system = mb.Compound()

# create a pattern of points to fill a disk
# patterns are generated between 0 and 1,
# and thus need to be scaled to provide appropriate spacing
pattern_disk = mb.DiskPattern(50)
pattern_disk.scale(5)

# now clone the polymer and move it to the points in the pattern
```

```
for pos in pattern_disk:
    current_polymer = mb.clone(polymer)
    current_polymer.translate(pos)
    system.add(current_polymer)

system.visualize()
```

Other patterns can be used, e.g., the `Grid3DPattern`. We can also use the rotation commands to randomize the orientation.

```
import random

comp = mb.load('C', smiles=True)
polymer = Polymer()

polymer.add_monomer(compound=comp,
                    indices=[1, -2],
                    separation=.15,
                    replace=True)

polymer.build(n=10, sequence='A')

system = mb.Compound()
polymer.rotate(np.pi/2, [1, 0, 0])

pattern_disk = mb.Grid3DPattern(5, 5, 5)
pattern_disk.scale(8.0)

for pos in pattern_disk:
    current_polymer = mb.clone(polymer)
    for around in [(1, 0, 0), (0, 1, 0), (0, 0, 1)]:  # rotate around x, y, and z
        current_polymer.rotate(random.uniform(0, np.pi), around)
    current_polymer.translate(pos)
    system.add(current_polymer)

system.visualize()
```

`mBuild` also provides an interface to `PACKMOL`, allowing the creation of a randomized configuration.

```
comp = mb.load('C', smiles=True) # mBuild compound of the monomer unit
polymer = Polymer()

polymer.add_monomer(compound=comp,
                    indices=[1, -2],
                    separation=.15,
                    replace=True)

polymer.build(n=5, sequence='A')

system = mb.fill_box(polymer, n_compounds=100, overlap=1.5, box=[10,10,10])
system.visualize()
```

Variations

Rather than a linear chain, the `Polymer` class we wrote can be easily changed such that small perturbations are given to each port. To avoid accumulation of deviations from the equilibrium angle, we will clone an unperturbed monomer each time (i.e., `monomer_proto`) before applying a random variation.

We also define a variable `delta`, which will control the maximum amount of perturbation. Note that large values of `delta` may result in the chain overlapping itself, as `mBuild` does not currently include routines to exclude such overlaps.

```python
import mbuild as mb

import random

class AlkanePolymer(mb.Compound):
    def __init__(self, chain_length=1, delta=0):
        super(AlkanePolymer, self).__init__()
        monomer_proto = CH2()
        last_monomer = CH2()
        last_monomer['down'].rotate(random.uniform(-delta,delta), [1, 0, 0])
        last_monomer['down'].rotate(random.uniform(-delta,delta), [0, 1, 0])
        self.add(last_monomer)
        for i in range(chain_length-1):
            current_monomer = mb.clone(monomer_proto)
            current_monomer['down'].rotate(random.uniform(-delta,delta), [1, 0,
 ↪0])
            current_monomer['down'].rotate(random.uniform(-delta,delta), [0, 1,
 ↪0])
            mb.force_overlap(move_this=current_monomer,
                             from_positions=current_monomer['up'],
                             to_positions=last_monomer['down'])
            self.add(current_monomer)
            last_monomer=current_monomer

polymer = AlkanePolymer(chain_length = 200, delta=0.4)
polymer.visualize()
```

Lattice Tutorial

The following notebook provides a thorough walkthrough to using the `Lattice` class to build up crystal systems.

Lattice Functionality

- **Variable-dimension crystal structures**
  - `Lattice` supports the dimensionality of `mBuild`, which means that the systems can be in 1D, 2D, or 3D. Replace the necessary vector components with 0 to emulate the dimensionality of interest.

- **Multicomponent crystals**
  - `Lattice` can support an indefinite amount of lattice points in its data structure.
  - The *repeat* cell can be as large as the user defines useful.
  - The components that occupy the lattice points are `mbuild.Compound` objects.

* This allows the user to build any system since compounds are only a representation of matter in this design.

* Molecular crystals, coarse grained, atomic, even alloy crystal structures are all supported

- **Triclinic Lattices**

  – With full support for triclinic lattices, any crystal structure can technically be developed.

  – Either the user can provide the lattice parameters, or if they know the vectors that span the unit cell, that can also be provided.

- **Generation of lattice structure from crystallographic index file** (CIF)[60] **formats**

  – Please also see the *Load files* section for other ways to load files.

- *IN PROGRESS* **Template recipes to generate common crystal structures (FCC, BCC, HEX, etc)**

  – This is currently being developed and will be released relatively shortly

  – To generate these structures currently, the user needs to know the lattice parameters or lattice vectors that define these units.

Lattice Data Structure Introduction

Below we will explore the relevant data structures that are attributes of the `Lattice` class. This information will be essential to build desired crystal structures.

To begin, we will call the python `help()` method to observe the parameters and attributes of the `Lattice` class.

```
import mbuild
help(mbuild.Lattice)
```

As we can see, there are quite a few attributes and parameters that make up this class. There are also a lot of inline examples as well. If you ever get stuck, remember to use the python built-in `help()` method!

- **Lattice.lattice_spacing**

  This data structure is a (3,) array that details the lengths of the repeat cell for the crystal. You can either use a `numpy` array object, or simply pass in a list and `Lattice` will handle the rest. Remember that `mBuild`'s units of length are in nanometers [nm]. You must pass in all three lengths, even if they are all equivalent. These are the lattice parameters a, b, c when viewing crystallographic information.

  For Example:

  ```
  lattice_spacing = [0.5, 0.5, 0.5]
  ```

- **Lattice.lattice_vectors**

  `lattice_vectors` is a 3x3 array that defines the vectors that encapsulate the repeat cell. This is an optional value that the user can pass in to define the cell. Either this must be passed in, or the 3 Bravais angles of the cell from the lattice parameters must be provided. If neither is passed in, the default value are the vectors that encase a cubic lattice.

---

[60] https://www.iucr.org/resources/cif/documentation

Note: Most users will **not** have to use these to build their lattice structure of interest. It will usually be easier for the users to provide the 3 Bravais angles instead. If the user then wants the vectors, the `Lattice` object will calculate them for the user.

For example: Cubic Cell

```
lattice_vectors = [[1, 0, 0], [0, 1, 0], [0, 0, 1]]
```

- **Lattice.angles**

  `angles` is a (3,) array that defines the three Bravais angles of the lattice. Commonly referred to as α, β, γ in the definition of the lattice parameters.

  For example: Cubic Cell

```
angles = [90, 90, 90]
```

- **Lattice.lattice_points**

  `lattice_points` can be the most common source of confusion when creating a crystal structure. In crystallographic terms, this is the minimum basis set of points in space that define where the points in the lattice exist. This requires that the user does not over define the system.

Note: MIT's OpenCourseWare has an excellent PDF for more information here[61]

The other tricky issue that can come up is the data structure itself. `lattice_points` is a dictionary where the `dict.key` items are the `string` id's for each basis point. The `dict.values` items are a nested list of fractional coordinates of the unique lattice points in the cell. If you have the same `Compound` at multiple lattice_points, it is easier to put all those coordinates in a nested list under the same `key` value. Two examples will be given below, both FCC unit cells, one with all the same id, and one with unique ids for each lattice_point.

For Example: FCC All Unique

```
lattice_points = {'A' : [[0, 0, 0]],
                  'B' : [[0.5, 0.5, 0]],
                  'C' : [[0.5, 0, 0.5]],
                  'D' : [[0, 0.5, 0.5]]
                  }
```

For Example: FCC All Same

```
lattice_points = {'A' : [[0, 0, 0], [0.5, 0.5, 0], [0.5, 0, 0.5], [0, 0.5, 0.
 →5]] }
```

---

[61] https://ocw.mit.edu/courses/earth-atmospheric-and-planetary-sciences/12-108-structure-of-earth-materials-fall-2004/lecture-notes/lec7.pdf

Lattice Public Methods

The `Lattice` class also contains methods that are responsible for applying `Compounds` to the lattice points, with user defined cell replications in the x, y, and z directions.

- **Lattice.populate(compound_dict=None, x=1, y=1, z=1)**

    This method uses the `Lattice` object to place `Compounds` at the specified `lattice_points`. There are 4 optional inputs for this class.

    - `compound_dict` inputs another dictionary that defines a relationship between the `lattice_points` and the `Compounds` that the user wants to populate the lattice with. The `dict.keys` of this dictionary must be the same as the `keys` in the `lattice_points` dictionary. However, for the `dict.items` in this case, the `Compound` that the user wants to place at that lattice point(s) will be used. An example will use the FCC examples from above. They have been copied below:

        For Example: FCC All Unique

        ```
        lattice_points = {'A' : [[0, 0, 0]],
                          'B' : [[0.5, 0.5, 0]],
                          'C' : [[0.5, 0, 0.5]],
                          'D' : [[0, 0.5, 0.5]]
                          }

        # compound dictionary
        a = mbuild.Compound(name='A')
        b = mbuild.Compound(name='B')
        c = mbuild.Compound(name='C')
        d = mbuild.Compound(name='D')
        compound_dict = {'A' : a, 'B' : b, 'C' : c, 'D' : d}
        ```

        For Example: FCC All Same

        ```
        lattice_points = {'A' : [[0, 0, 0], [0.5, 0.5, 0], [0.5, 0, 0.5],␣
         ↪[0, 0.5, 0.5]] }

        # compound dictionary
        a = mbuild.Compound(name='A')
        compound_dict = {'A' : a}
        ```

Example Lattice Systems

Below contains some examples of homogeneous and heterogeneous 2D and 3D lattice structures using the `Lattice` class.

Simple Cubic (SC)

- Polonium

```python
import mbuild as mb
import numpy as np

# define all necessary lattice parameters
spacings = [0.3359, 0.3359, 0.3359]
angles = [90, 90, 90]
points = [[0, 0, 0]]

# define lattice object
sc_lattice = mb.Lattice(lattice_spacing=spacings, angles=angles, lattice_points={
 →'Po' : points})

# define Polonium Compound
po = mb.Compound(name='Po')

# populate lattice with compounds
po_lattice = sc_lattice.populate(compound_dict={'Po' : po}, x=2, y=2, z=2)

# visualize
po_lattice.visualize()
```


Fig. 4: **Polonium simple cubic (SC) structure**

Body-centered Cubic (BCC)

- CsCl

```python
import mbuild as mb
import numpy as np

# define all necessary lattice parameters
spacings = [0.4123, 0.4123, 0.4123]
angles = [90, 90, 90]
point1 = [[0, 0, 0]]
point2 = [[0.5, 0.5, 0.5]]

# define lattice object
bcc_lattice = mb.Lattice(lattice_spacing=spacings, angles=angles, lattice_points={
 ↪'A' : point1, 'B' : point2})

# define Compounds
cl = mb.Compound(name='Cl')
cs = mb.Compound(name='Cs')

# populate lattice with compounds
cscl_lattice = bcc_lattice.populate(compound_dict={'A' : cl, 'B' : cs}, x=2, y=2,
 ↪z=2)

# visualize
cscl_lattice.visualize()
```


Fig. 5: **CsCl body-centered cubic (BCC) structure**

Face-centered Cubic (FCC)

- Cu

```python
import mbuild as mb
import numpy as np

# define all necessary lattice parameters
spacings = [0.36149, 0.36149, 0.36149]
angles = [90, 90, 90]
points = [[0, 0, 0], [0.5, 0.5, 0], [0.5, 0, 0.5], [0, 0.5, 0.5]]
```

```python
# define lattice object
fcc_lattice = mb.Lattice(lattice_spacing=spacings, angles=angles, lattice_points={
→'A' : points})

# define Compound
cu = mb.Compound(name='Cu')

# populate lattice with compounds
cu_lattice = fcc_lattice.populate(compound_dict={'A' : cu}, x=2, y=2, z=2)

# visualize
cu_lattice.visualize()
```
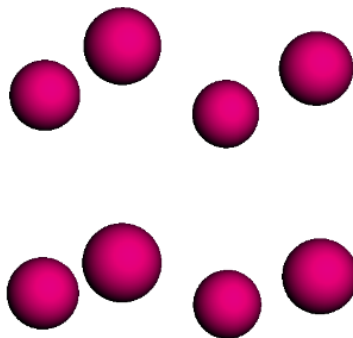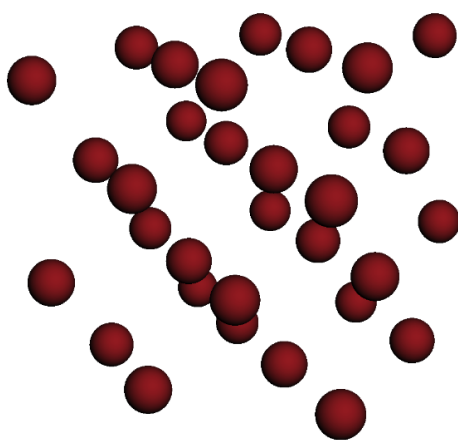

Fig. 6: **Cu face-centered cubic (FCC) structure**

Diamond (Cubic)

- Si

```python
import mbuild as mb
import numpy as np

# define all necessary lattice parameters
spacings = [0.54309, 0.54309, 0.54309]
angles = [90, 90, 90]
points = [[0, 0, 0], [0.5, 0.5, 0], [0.5, 0, 0.5], [0, 0.5, 0.5],
          [0.25, 0.25, 0.75], [0.25, 0.75, 0.25], [0.75, 0.25, 0.25], [0.75, 0.75,
→ 0.75]]

# define lattice object
diamond_lattice = mb.Lattice(lattice_spacing=spacings, angles=angles, lattice_
→points={'A' : points})

# define Compound
si = mb.Compound(name='Si')

# populate lattice with compounds
```

```
si_lattice = diamond_lattice.populate(compound_dict={'A' : si}, x=2, y=2, z=2)

# visualize
si_lattice.visualize()
```
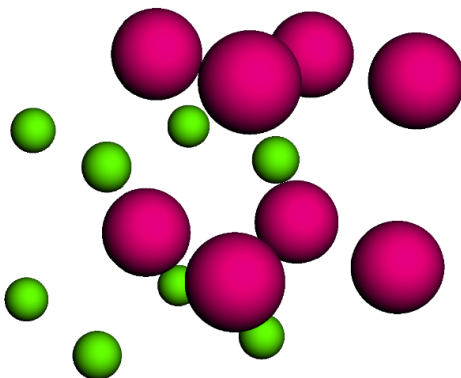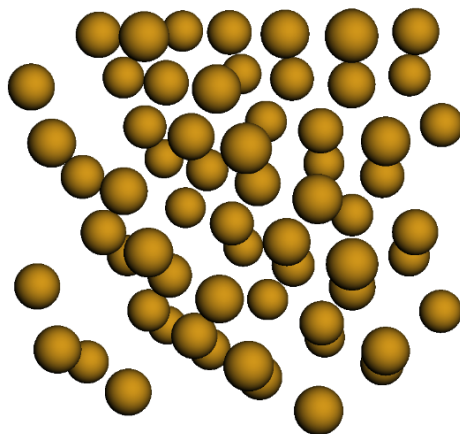

Fig. 7: **Si diamond (Cubic) structure**

Graphene (2D)

- C

```
import mbuild as mb
import numpy as np


# define all necessary lattice parameters
spacings = [0.246, 0.246, 0.335]
angles = [90, 90, 120]
points = [[0, 0, 0], [1/3, 2/3, 0]]

# define lattice object
graphene_lattice = mb.Lattice(lattice_spacing=spacings, angles=angles, lattice_
↪points={'A' : points})

# define Compound
c = mb.Compound(name='C')

# populate lattice with compounds
graphene = graphene_lattice.populate(compound_dict={'A' : c}, x=5, y=5, z=1)

# visualize
graphene.visualize()
```
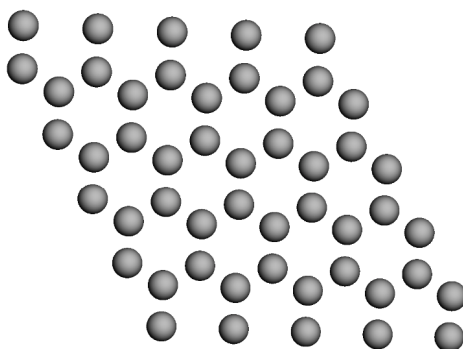
Fig. 8: **Graphene (2D) structure**

1.6 Recipe Development

There may be cases where your `Compounds` and/or building scripts can be generalized to support a broad range of systems. Such objects would be a valuable resource for many researchers, and might justify development of a Python package that could be distributed to the community.

`mBuild` has been developed with this in mind, in the form of a plug-in system. Detailed below are the specifications of this system, how to convert an existing Python project into an mBuild-discoverable plug-in, and an example.

Entry Points

The basis of the plug-in system in mBuild is the setuptools.entry_points package[62]. This allows other packages to register themselves with the `entry_point` group we defined in mBuild, so they are accessible through the `mbuild.recipes` location. Imagine you have a class named `my_foo` that inherits from `mb.Compound`. It is currently inside of a project `my_project` and is accessed via a direct import, i.e. `from my_project import my_foo`. You can register this class as an entry point associated with `mbuild.recipes`. It will then be accessible from inside mBuild as a plug-in via `mbuild.recipes.my_foo` and a direct import will be unncessary. The call `import mbuild` discovers all plug-ins that fit the `entry_point` group specification and makes them available under `mbuild.recipes`.

Registering a Recipe

Here we consider the case that a user already has a Python project set up with a structure similar to the layout below.

This project can be found here[63].

```
mbuild_fcc
├── LICENSE
├── README.md
├── mbuild_fcc
│   ├── mbuild_fcc.py
│   └── tests
│       ├── __init__.py
│       └── test_fcc.py
└── setup.py
```

[62] https://packaging.python.org/guides/creating-and-discovering-plugins/#using-package-metadata
[63] https://github.com/justinGilmer/mbuild-fcc

49

The two important files for the user to convert their `mBuild` plug-in to a discoverable plug-in are `setup.py` and `mbuild_fcc.py`.

To begin, lets first inspect the `mbuild_fcc.py` file, a shortened snippet is below.

```python
import mbuild


class FCC(mbuild.Compound):
    """Create a mBuild Compound with a repeating unit of the FCC unit cell.

    ... (shortened for viewability)

    """

    def __init__(self, lattice_spacing=None, compound_to_add=None, x=1, y=1, z=1):
        super(FCC, self).__init__()

        # ... (shortened for viewability)

if __name__ == "__main__":
    au_fcc_lattice = FCC(lattice_spacing=0.40782,
                         compound_to_add=mbuild.Compound(name="Au"),
                         x=5, y=5, z=1)
    print(au_fcc_lattice)
```

There are two notable lines in this file that we need to focus on when developing this as a plug-in for mBuild.

The first is the import statement `import mbuild`. We must make sure that mbuild is installed since we are inheriting from `mbuild.Compound`. When you decide to distribute your plug-in, the dependencies must be listed.

The second is to select the name of the plug-in itself. It is considered good practice to name it the name of your `class`. In this case, we will name the plug-in `FCC`.

The last step is to edit the `setup.py` file such that the plug-in can be registered under the entry_point group `mbuild.plugins`.

```python
from setuptools import setup

setup(
    ...
    entry_points={ "mbuild.plugins":[ "FCC = mbuild_fcc.mbuild_fcc:FCC"]},
    ...
)
```

The important section is the `entry_points` argument. Here we define the entry_point group we want to register with: `"mbuild.plugins"`. Finally, we tell Python what name to use when accessing this plug-in. Earlier, we decided to call it `FCC`. This is denoted here by the name before the assignment operator `FCC =`. Next, we pass the location of the file with our plug-in: `mbuild_fcc.mbuild_fcc` as if we were located at the `setup.py` file. Then, we provide the name of the class within that Python file we want to make discoverable :`FCC`.

Since the `setup.py` file is located in the top folder of the python project, the first `mbuild_fcc` is the name of the folder, and the second is the name of the python file. The colon (:) is used when accessing the class that is in the python file itself.

Putting it all together

Finally, we have `FCC = mbuild_fcc.mbuild_fcc:FCC`.

To test this feature, you should clone the `mbuild-fcc` project listed above.

`git clone https://github.com/justinGilmer/mbuild-fcc`

Make sure you have mBuild installed, then run the command below after changing into the `mbuild-fcc` directory.

`cd mbuild-fcc`

`pip install -e .`

Note that this command will install this example from source in an editable format.

Trying it Out

To test that you set up your plug-in correctly, try importing mBuild:

`import mbuild`

If you do not receive error messages, your plug-in should be discoverable!

`help(mbuild.recipes.FCC)` `

1.7 Data Structures

The primary building blocks in an mBuild hierarchy inherit from the *`mbuild.Compound`* class. `Compounds` maintain an ordered set of `children` which are other `Compounds`. In addition, an independent, ordered dictionary of `labels` is maintained through which users can reference any other `Compound` in the hierarchy via descriptive strings. Every `Compound` knows its parent `Compound`, one step up in the hierarchy, and knows which `Compounds` reference it in their `labels`. *`mbuild.Port`* is a special type of `Compound` which are used internally to connect different `Compounds` using the equivalence transformations described below.

`Compounds` at the bottom of an mBuild hierarchy, the leaves of the tree, are referred to as `Particles` and can be instantiated as `foo = mbuild.Particle(name='bar')`. Note however, that this merely serves to illustrate that this `Compound` is at the bottom of the hierarchy; `Particle` is simply an alias for `Compound` which can be used to clarify the intended role of an object you are creating. The method *`mbuild.Compound.particles()`* traverses the hierarchy to the bottom and yields those `Compounds`. *`mbuild.Compound.root()`* returns the compound at the top of the hierarchy.

Compound

class mbuild.**Compound**(*subcompounds=None, name=None, pos=None, mass=None, charge=None, periodicity=None, box=None, element=None, port_particle=False*)

A building block in the mBuild hierarchy.

Compound is the superclass of all composite building blocks in the mBuild hierarchy. That is, all composite building blocks must inherit from compound, either directly or indirectly. The design of Compound follows the Composite design pattern:

```
@book{DesignPatterns,
    author = "Gamma, Erich and Helm, Richard and Johnson, Ralph and
    Vlissides, John M.",
    title = "Design Patterns",
```

```
    subtitle = "Elements of Reusable Object-Oriented Software",
    year = "1995",
    publisher = "Addison-Wesley",
    note = "p. 395",
    ISBN = "0-201-63361-2",
}
```

with Compound being the composite, and Particle playing the role of the primitive (leaf) part, where Particle is in fact simply an alias to the Compound class.

Compound maintains a list of children (other Compounds contained within), and provides a means to tag the children with labels, so that the compounds can be easily looked up later. Labels may also point to objects outside the Compound's containment hierarchy. Compound has built-in support for copying and deepcopying Compound hierarchies, enumerating particles or bonds in the hierarchy, proximity based searches, visualization, I/O operations, and a number of other convenience methods.

Parameters

subcompounds
> [mb.Compound or list of mb.Compound, optional, default=None] One or more compounds to be added to self.

name
> [str, optional, default=self.__class__.__name__] The type of Compound.

pos
> [np.ndarray, shape=(3,), dtype=float, optional, default=[0, 0, 0]] The position of the Compound in Cartestian space

mass
> [float, optional, default=None] The mass of the compound. If none is set, then will try to infer the mass from a compound's element attribute. If neither *mass* or *element* are specified, then the mass will be None.

charge
> [float, optional, default=0.0] Currently not used. Likely removed in next release.

periodicity
> [tuple of bools, length=3, optional, default=None] Whether the Compound is periodic in the x, y, and z directions. If None is provided, the periodicity is set to (False, False, False) which is non-periodic in all directions.

port_particle
> [bool, optional, default=False] Whether or not this Compound is part of a Port

box
> [mb.Box, optional] The simulation box containing the compound. Also accounts for the periodicity. Defaults to None which is treated as non-periodic.

element: str, optional, default=None
> The one or two character element symbol

Attributes

bond_graph
> [mb.BondGraph] Graph-like object that stores bond information for this Compound

children

[list] Contains all children (other Compounds).

labels

[OrderedDict] Labels to Compound/Atom mappings. These do not necessarily need not be in self.children.

parent

[mb.Compound] The parent Compound that contains this part. Can be None if this compound is the root of the containment hierarchy.

referrers

[set] Other compounds that reference this part with labels.

rigid_id

[int, default=None] Get the rigid_id of the Compound.

boundingbox

[mb.Box] The bounds (xmin, xmax, ymin, ymax, zmin, zmax) of particles in Compound

center

Get the cartesian center of the Compound based on its Particles.

contains_rigid

Return True if the Compound contains rigid bodies.

mass

Return the total mass of a compound.

max_rigid_id

Return the maximum rigid body ID contained in the Compound.

n_particles

Return the number of Particles in the Compound.

n_bonds

Return the total number of bonds in the Compound.

root

Get the Compound at the top of self's hierarchy.

xyz

Return all particle coordinates in this compound.

xyz_with_ports

Return all particle coordinates in this compound including ports.

add(*new_child*, *label=None*, *containment=True*, *replace=False*, *inherit_periodicity=None*, *inherit_box=False*, *reset_rigid_ids=True*, *check_box_size=True*)

Add a part to the Compound.

Note:

This does not necessarily add the part to self.children but may instead be used to add a reference to the part to self.labels. See 'containment' argument.

Parameters

new_child

[mb.Compound or list-like of mb.Compound] The object(s) to be added to this Compound.

label

[str, or list-like of str, optional, default None] A descriptive string for the part; if a list, must be the same length/shape as new_child.

containment
: [bool, optional, default=True] Add the part to self.children.

replace
: [bool, optional, default=True] Replace the label if it already exists.

inherit_periodicity
: [bool, optional, default=True] Replace the periodicity of self with the periodicity of the Compound being added

inherit_box: bool, optional, default=False
: Replace the box of self with the box of the Compound being added

reset_rigid_ids
: [bool, optional, default=True] If the Compound to be added contains rigid bodies, reset the rigid_ids such that values remain distinct from rigid_ids already present in *self*. Can be set to False if attempting to add Compounds to an existing rigid body.

check_box_size
: [bool, optional, default=True] Checks and warns if compound box is smaller than its bounding box after adding new_child.

add_bond(*particle_pair*, *bond_order=None*)

    Add a bond between two Particles.

    Parameters

        particle_pair
:   [indexable object, length=2, dtype=mb.Compound] The pair of Particles to add a bond between

        bond_order
:   [float, optional, default=None] Bond order of the bond. Available options include "default", "single", "double", "triple", "aromatic" or "unspecified"

all_ports()

    Return all Ports referenced by this Compound and its successors.

    Returns

        list of mb.Compound
:   A list of all Ports referenced by this Compound and its successors

ancestors()

    Generate all ancestors of the Compound recursively.

    Yields

        mb.Compound
:   The next Compound above self in the hierarchy

available_ports()

    Return all unoccupied Ports referenced by this Compound.

    Returns

        list of mb.Compound
:   A list of all unoccupied ports referenced by the Compound

bonds(*return_bond_order=False*)

    Return all bonds in the Compound and sub-Compounds.

    Parameters

return_bond_order

[bool, optional, default=False] Return the bond order of the bond as the 3rd argument in the tuple. bond order is returned as a dictionary with 'bo' as the key. If bond order is not set, the value will be set to 'default'.

Yields

tuple of mb.Compound

The next bond in the Compound

See also:

bond_graph.edges_iter

Iterates over all edges in a BondGraph

Compound.n_bonds

Returns the total number of bonds in the Compound and sub-Compounds

property box

Get the box of the Compound.

Ports cannot have a box.

property center

Get the cartesian center of the Compound based on its Particles.

Returns

np.ndarray, shape=(3,), dtype=float

The cartesian center of the Compound based on its Particles

property charge

Return the total charge of a compound.

If the compound contains children compouds, the total charge of all children compounds is returned.

If the charge of a particle has not been explicitly set then the particle's charge is None, and are not used when calculating the total charge.

condense(*inplace=True*)

Condense the hierarchical structure of the Compound to the level of molecules.

Modify the mBuild Compound to become a Compound with 3 distinct levels in the hierarchy. The top level container (self), contains molecules (i.e., connected Compounds) and the third level represents Particles (i.e., Compounds with no children). If the system contains a Particle(s) without any connections to other Compounds, it will appear in the 2nd level (with the top level self as a parent).

property contains_rigid

Return True if the Compound contains rigid bodies.

If the Compound contains any particle with a rigid_id != None then contains_rigid will return True. If the Compound has no children (i.e. the Compound resides at the bottom of the containment hierarchy) then contains_rigid will return False.

Returns

bool,

True if the Compound contains any particle with a rigid_id != None

Notes

The private variable '_check_if_contains_rigid_bodies' is used to help cache the status of 'contains_rigid'. If '_check_if_contains_rigid_bodies' is False, then the rigid body containment of the Compound has not changed, and the particle tree is not traversed, boosting performance.

`direct_bonds()`

Return a list of particles that this particle bonds to.

Returns

List of mb.Compound

See also:

`bond_graph.edges_iter`
Iterations over all edges in a BondGraph

`Compound.n_direct_bonds`
Returns the number of bonds a particle contains

`property element`

Get the element of the Compound.

`energy_minimize`(*forcefield='UFF'*, *steps=1000*, *shift_com=True*, *anchor=None*, ***kwargs*)

Perform an energy minimization on a Compound.

Default behavior utilizes Open Babel[64] to perform an energy minimization/geometry optimization on a Compound by applying a generic force field

Can also utilize OpenMM[65] to energy minimize after atomtyping a Compound using Foyer[66] to apply a forcefield XML file that contains valid SMARTS strings.

This function is primarily intended to be used on smaller components, with sizes on the order of 10's to 100's of particles, as the energy minimization scales poorly with the number of particles.

Parameters

steps
[int, optional, default=1000] The number of optimization iterations

forcefield
[str, optional, default='UFF'] The generic force field to apply to the Compound for minimization. Valid options are 'MMFF94', 'MMFF94s', ''UFF', 'GAFF', 'Ghemical'. Please refer to the Open Babel documentation[67] when considering your choice of force field. Utilizing OpenMM for energy minimization requires a forcefield XML file with valid SMARTS strings. Please refer to OpenMM docs[68] for more information.

shift_com
[bool, optional, default=True] If True, the energy-minimized Compound is translated such that the center-of-mass is unchanged relative to the initial configuration.

anchor
[Compound, optional, default=None] Translates the energy-minimized Compound such that the position of the anchor Compound is unchanged relative to the initial configuration.

Other Parameters

algorithm

[str, optional, default='cg'] The energy minimization algorithm. Valid options are 'steep', 'cg', and 'md', corresponding to steepest descent, conjugate gradient, and equilibrium molecular dynamics respectively. For _energy_minimize_openbabel

fixed_compounds

[Compound, optional, default=None] An individual Compound or list of Compounds that will have their position fixed during energy minimization. Note, positions are fixed using a restraining potential and thus may change slightly. Position fixing will apply to all Particles (i.e., atoms) that exist in the Compound and to particles in any subsequent sub-Compounds. By default x,y, and z position is fixed. This can be toggled by instead passing a list containing the Compound and an list or tuple of bool values corresponding to x,y and z; e.g., [Compound, (True, True, False)] will fix the x and y position but allow z to be free. For _energy_minimize_openbabel

ignore_compounds: Compound, optional, default=None

An individual compound or list of Compounds whose underlying particles will have their positions fixed and not interact with other atoms via the specified force field during the energy minimization process. Note, a restraining potential used and thus absolute position may vary as a result of the energy minimization process. Interactions of these ignored atoms can be specified by the user, e.g., by explicitly setting a distance constraint. For _energy_minimize_openbabel

distance_constraints: list, optional, default=None

A list containing a pair of Compounds as a tuple or list and a float value specifying the target distance between the two Compounds, e.g.,: [(compound1, compound2), distance]. To specify more than one constraint, pass constraints as a 2D list, e.g.,: [[(compound1, compound2), distance1], [(compound3, compound4), distance2]]. Note, Compounds specified here must represent individual point particles. For _energy_minimize_openbabel

constraint_factor: float, optional, default=50000.0

Harmonic springs are used to constrain distances and fix atom positions, where the resulting energy associated with the spring is scaled by the constraint_factor; the energy of this spring is considering during the minimization. As such, very small values of the constraint_factor may result in an energy minimized state that does not adequately restrain the distance/position of atoms. For _energy_minimize_openbabel

scale_bonds

[float, optional, default=1] Scales the bond force constant (1 is completely on). For _energy_minimize_openmm

scale_angles

[float, optional, default=1] Scales the angle force constant (1 is completely on) For _energy_minimize_openmm

scale_torsions

[float, optional, default=1] Scales the torsional force constants (1 is completely on) For _energy_minimize_openmm Note: Only Ryckaert-Bellemans style torsions are currently supported

scale_nonbonded

[float, optional, default=1] Scales epsilon (1 is completely on) For _energy_minimize_openmm

constraints

[str, optional, default="AllBonds"] Specify constraints on the molecule to minimize, options are: None, "HBonds", "AllBonds", "HAngles" For _energy_minimize_openmm

References

If using _energy_minimize_openmm(), please cite:

If using _energy_minimize_openbabel(), please cite:

If using the 'MMFF94' force field please also cite the following:

If using the 'MMFF94s' force field please cite the above along with:

If using the 'UFF' force field please cite the following:

If using the 'GAFF' force field please cite the following:

If using the 'Ghemical' force field please cite the following:

[Eastman2013], [OBoyle2011], [OpenBabel], [Halgren1996a], [Halgren1996b], [Halgren1996c], [Halgren1996d], [Halgren1996e], [Halgren1999], [Rappe1992], [Wang2004], [Hassinen2001]

flatten(*inplace=True*)

Flatten the hierarchical structure of the Compound.

Modify the mBuild Compound to become a Compound where there is a single container (self) that contains all the particles.

freud_generate_bonds(*name_a*, *name_b*, *dmin*, *dmax*)

Add Bonds between all pairs of types a/b within [dmin, dmax].

> **Parameters**
>
> > **name_a**
> > [str] The name of one of the Particles to be in each bond
> >
> > **name_b**
> > [str] The name of the other Particle to be in each bond
> >
> > **dmin**
> > [float] The minimum distance (in nm) between Particles for considering a bond
> >
> > **dmax**
> > [float] The maximum distance (in nm) between Particles for considering a bond

> **Notes**

This is an experimental feature and some behavior might change out of step of a standard development release.

from_gmso(*topology*, *coords_only=False*, *infer_hierarchy=True*)

Convert a GMSO Topology to mBuild Compound.

> **Returns**
>
> > **compound**
> > [mb.Compound]

from_parmed(*structure*, *coords_only=False*, *infer_hierarchy=True*)

    Extract atoms and bonds from a pmd.Structure.

    Will create sub-compounds for every chain if there is more than one and sub-sub-compounds for every residue.

    Parameters

        structure
            [pmd.Structure] The structure to load.

        coords_only
            [bool] Set preexisting atoms in compound to coordinates given by structure.

        infer_hierarchy
            [bool, optional, default=True] If true, infer compound hierarchy from chains and residues

from_pybel(*pybel_mol*, *use_element=True*, *coords_only=False*, *infer_hierarchy=True*, *ignore_box_warn=False*)

    Create a Compound from a Pybel.Molecule.

    Parameters

        pybel_mol: pybel.Molecule
        use_element
            [bool, default True] If True, construct mb Particles based on the pybel Atom's element. If False, construcs mb Particles based on the pybel Atom's type

        coords_only
             [bool, default False] Set preexisting atoms in compound to coordinates given by structure. Note: Not yet implemented, included only for parity with other conversion functions

        infer_hierarchy
             [bool, optional, default=True] If True, infer hierarchy from residues

        ignore_box_warn
             [bool, optional, default=False] If True, ignore warning if no box is present.

    See also:

    `mbuild.conversion.from_pybel`

from_trajectory(*traj*, *frame=-1*, *coords_only=False*, *infer_hierarchy=True*)

    Extract atoms and bonds from a md.Trajectory.

    Will create sub-compounds for every chain if there is more than one and sub-sub-compounds for every residue.

    Parameters

        traj
            [mdtraj.Trajectory] The trajectory to load.

        frame
            [int, optional, default=-1 (last)] The frame to take coordinates from.

        coords_only
            [bool, optional, default=False] Only read coordinate information

        infer_hierarchy
            [bool, optional, default=True] If True, infer compound hierarchy from chains and residues

See also:

`mbuild.conversion.from_trajectory`

`generate_bonds`(*name_a, name_b, dmin, dmax*)

Add Bonds between all pairs of types a/b within [dmin, dmax].

> **Parameters**
>
> > **name_a**
> > [str] The name of one of the Particles to be in each bond
> >
> > **name_b**
> > [str] The name of the other Particle to be in each bond
> >
> > **dmin**
> > [float] The minimum distance (in nm) between Particles for considering a bond
> >
> > **dmax**
> > [float] The maximum distance (in nm) between Particles for considering a bond

`get_boundingbox`(*pad_box=None*)

Compute the bounding box of the compound.

Compute and store the rectangular bounding box of the Compound.

> **Parameters**
>
> > **pad_box: Sequence, optional, default=None**
> > Pad all lengths or a list of lengths by a specified amount in nm. Acceptable values are:
> >
> > - A single float: apply this pad value to all 3 box lengths.
> > - A sequence of length 1: apply this pad value to all 3 box lengths.
> > - A sequence of length 3: apply these pad values to the a, b, c box lengths.
>
> **Returns**
>
> > **mb.Box**
> > The bounding box for this Compound.

> **Notes**

Triclinic bounding boxes are supported, but only for Compounds that are generated from mb.Lattice's and the resulting mb.Lattice.populate method

`get_smiles`()

Get SMILES string for compound.

Bond order is guessed with pybel and may lead to incorrect SMILES strings.

> **Returns**
>
> > **smiles_string: str**

`is_independent`()

Return True if there is no bond between particles of the Compound to an external Compound.

label_rigid_bodies(*discrete_bodies=None*, *rigid_particles=None*)

Designate which Compounds should be treated as rigid bodies.

If no arguments are provided, this function will treat the compound as a single rigid body by providing all particles in *self* with the same rigid_id. If *discrete_bodies* is not None, each instance of a Compound with a name found in *discrete_bodies* will be treated as a unique rigid body. If *rigid_particles* is not None, only Particles (Compounds at the bottom of the containment hierarchy) matching this name will be considered part of the rigid body.

> **Parameters**
>
> > **discrete_bodies**
> > [str or list of str, optional, default=None] Name(s) of Compound instances to be treated as unique rigid bodies. Compound instances matching this (these) name(s) will be provided with unique rigid_ids
> >
> > **rigid_particles**
> > [str or list of str, optional, default=None] Name(s) of Compound instances at the bottom of the containment hierarchy (Particles) to be included in rigid bodies. Only Particles matching this (these) name(s) will have their rigid_ids altered to match the rigid body number.

> **Examples**

Creating a rigid benzene

```
>>> import mbuild as mb
>>> from mbuild.utils.io import get_fn
>>> benzene = mb.load(get_fn('benzene.mol2'))
>>> benzene.label_rigid_bodies()
```

Creating a semi-rigid benzene, where only the carbons are treated as a rigid body

```
>>> import mbuild as mb
>>> from mbuild.utils.io import get_fn
>>> benzene = mb.load(get_fn('benzene.mol2'))
>>> benzene.label_rigid_bodies(rigid_particles='C')
```

Create a box of rigid benzenes, where each benzene has a unique rigid body ID.

```
>>> import mbuild as mb
>>> from mbuild.utils.io import get_fn
>>> benzene = mb.load(get_fn('benzene.mol2'))
>>> benzene.name = 'Benzene'
>>> filled = mb.fill_box(benzene,
...                      n_compounds=10,
...                      box=[0, 0, 0, 4, 4, 4])
>>> filled.label_rigid_bodies(distinct_bodies='Benzene')
```

Create a box of semi-rigid benzenes, where each benzene has a unique rigid body ID and only the carbon portion is treated as rigid.

```
>>> import mbuild as mb
>>> from mbuild.utils.io import get_fn
>>> benzene = mb.load(get_fn('benzene.mol2'))
>>> benzene.name = 'Benzene'
>>> filled = mb.fill_box(benzene,
```

```
...                                     n_compounds=10,
...                                     box=[0, 0, 0, 4, 4, 4])
>>> filled.label_rigid_bodies(distinct_bodies='Benzene',
...                             rigid_particles='C')
```

property mass

Return the total mass of a compound.

If the compound contains children compouds, the total mass of all children compounds is returned. If the compound contains element information (Compound.element) then the mass is inferred from the elemental mass. If Compound.mass has been set explicitly, then it will override the mass inferred from Compound.element. If neither of a Compound's element or mass attributes have been set, then a mass of zero is returned.

property max_rigid_id

Return the maximum rigid body ID contained in the Compound.

This is usually used by compound.root to determine the maximum rigid_id in the containment hierarchy.

> **Returns**
>
> > **int or None**
> > The maximum rigid body ID contained in the Compound. If no rigid body IDs are found, None is returned

property maxs

Return the maximum x, y, z coordinate of any particle in this compound.

min_periodic_distance(*xyz0*, *xyz1*)

Vectorized distance calculation considering minimum image.

Only implemented for orthorhombic simulation boxes.

> **Parameters**
>
> > **xyz0**
> > [np.ndarray, shape=(3,), dtype=float] Coordinates of first point
> >
> > **xyz1**
> > [np.ndarray, shape=(3,), dtype=float] Coordinates of second point
>
> **Returns**
>
> > **float**
> > Vectorized distance between the two points following minimum image convention

property mins

Return the mimimum x, y, z coordinate of any particle in this compound.

property n_bonds

Return the total number of bonds in the Compound.

> **Returns**
>
> > **int**
> > The number of bonds in the Compound

property n_direct_bonds

Return the number of bonds a particle is directly involved in.

This method should only be used on on compounds at the bottom of their hierarchy (i.e. a particle).

> **Returns**
>
> > **int**
> > The number of compounds this compound is directly bonded to.

property n_particles

Return the number of Particles in the Compound.

> **Returns**
>
> > **int,**
> > The number of Particles in the Compound

particles(*include_ports=False*)

Return all Particles of the Compound.

> **Parameters**
>
> > **include_ports**
> > [bool, optional, default=False] Include port particles
>
> **Yields**
>
> > **mb.Compound**
> > The next Particle in the Compound

particles_by_element(*element*)

Return all Particles of the Compound with a specific element.

> **Parameters**
>
> > **name**
> > [str or ele.Element] element abbreviation or element
>
> **Yields**
>
> > **mb.Compound**
> > The next Particle in the Compound with the user-specified element

particles_by_name(*name*)

Return all Particles of the Compound with a specific name.

> **Parameters**
>
> > **name**
> > [str] Only particles with this name are returned
>
> **Yields**
>
> > **mb.Compound**
> > The next Particle in the Compound with the user-specified name

particles_in_range(*compound, dmax, max_particles=20, particle_kdtree=None,*
              particle_array=None)

Find particles within a specified range of another particle.

> **Parameters**
>
> > **compound**
> > [mb.Compound] Reference particle to find other particles in range of

dmax
[float] Maximum distance from 'compound' to look for Particles

max_particles
[int, optional, default=20] Maximum number of Particles to return

particle_kdtree
[mb.PeriodicKDTree, optional] KD-tree for looking up nearest neighbors. If not provided, a KD- tree will be generated from all Particles in self

particle_array
[np.ndarray, shape=(n,), dtype=mb.Compound, optional] Array of possible particles to consider for return. If not provided, this defaults to all Particles in self

Returns

np.ndarray, shape=(n,), dtype=mb.Compound
Particles in range of compound according to user-defined limits

See also:

`periodic_kdtree.PerioidicKDTree`
mBuild implementation of kd-trees

`scipy.spatial.kdtree`
Further details on kd-trees

property periodicity
Get the periodicity of the Compound.

property pos
Get the position of the Compound.

If the Compound contains children, returns the center.

The position of a Compound containing children can't be set.

print_hierarchy(*print_full=False*, *index=None*, *show_tree=True*)
Print the hierarchy of the Compound.

Parameters

print_full: bool, optional, default=False
The full hierarchy will be printed, rather than condensing compounds with identical topologies. Topologies are considered identical if they have the same name, contain the number and names of children, contain the same number and names of particles, and the same number of bonds.

index: int, optional, default=None
Print the branch of the first level of the hiearchy corresponding to the value specified by index. This only applies when print_full is True.

show_tree: bool, optional, default=True
If False, do not print the tree to the screen.

Returns

tree, treelib.tree.Tree, hierarchy of the compound as a tree

referenced_ports()
Return all Ports referenced by this Compound.

Returns

list of mb.Compound
A list of all ports referenced by the Compound

`remove`(*objs_to_remove*, *reset_labels=True*)

Remove children from the Compound cleanly.

Parameters

objs_to_remove
[mb.Compound or list of mb.Compound] The Compound(s) to be removed from self

reset_labels
[bool] If True, the Compound labels will be reset

`remove_bond`(*particle_pair*)

Delete a bond between a pair of Particles.

Parameters

particle_pair
[indexable object, length=2, dtype=mb.Compound] The pair of Particles to remove the bond between

`property rigid_id`

Get the rigid_id of the Compound.

`rigid_particles`(*rigid_id=None*)

Generate all particles in rigid bodies.

If a rigid_id is specified, then this function will only yield particles with a matching rigid_id.

Parameters

rigid_id
[int, optional] Include only particles with this rigid body ID

Yields

mb.Compound
The next particle with a rigid_id that is not None, or the next particle with a matching rigid_id if specified

`property root`

Get the Compound at the top of self's hierarchy.

Returns

mb.Compound
The Compound at the top of self's hierarchy

`rotate`(*theta*, *around*)

Rotate Compound around an arbitrary vector.

Parameters

theta
[float] The angle by which to rotate the Compound, in radians.

around
[np.ndarray, shape=(3,), dtype=float] The vector about which to rotate the Compound.

`rotate_dihedral`(*bond*, *phi*)

> Rotate a dihedral about a central bond.

> **Parameters**

>> **bond**
>> [indexable object, length=2, dtype=mb.Compound] The pair of bonded Particles in the central bond of the dihedral

>> **phi**
>> [float] The angle by which to rotate the dihedral, in radians.

`save`(*filename*, *include_ports=False*, *forcefield_name=None*, *forcefield_files=None*, *forcefield_debug=False*, *box=None*, *overwrite=False*, *residues=None*, *combining_rule='lorentz'*, *foyer_kwargs=None*, *parmed_kwargs=None*, ***kwargs*)

> Save the Compound to a file.

> **Parameters**

>> **filename**
>> [str] Filesystem path in which to save the trajectory. The extension or prefix will be parsed and control the format. Supported extensions: 'hoomdxml', 'gsd', 'gro', 'top', 'lammps', 'lmp', 'mcf', 'pdb', 'xyz', 'json', 'mol2', 'sdf', 'psf'. See parmed/structure.py for more information on savers.

>> **include_ports**
>> [bool, optional, default=False] Save ports contained within the compound.

>> **forcefield_files**
>> [str, optional, default=None] Apply a forcefield to the output file using a forcefield provided by the *foyer* package.

>> **forcefield_name**
>> [str, optional, default=None] Apply a named forcefield to the output file using the *foyer* package, e.g. 'oplsaa'. Foyer forcefields[69]

>> **forcefield_debug**
>> [bool, optional, default=False] Choose verbosity level when applying a forcefield through *foyer*. Specifically, when missing atom types in the forcefield xml file, determine if the warning is condensed or verbose.

>> **box**
>> [mb.Box, optional, default=self.boundingbox (with buffer)] Box information to be written to the output file. If 'None', a bounding box is used with 0.25nm buffers at each face to avoid overlapping atoms.

>> **overwrite**
>> [bool, optional, default=False] Overwrite if the filename already exists

>> **residues**
>> [str of list of str] Labels of residues in the Compound. Residues are assigned by checking against Compound.name.

>> **combining_rule**
>> [str, optional, default='lorentz'] Specify the combining rule for nonbonded interactions. Only relevant when the *foyer* package is used to apply a forcefield. Valid options are 'lorentz' and 'geometric', specifying Lorentz-Berthelot and geometric combining rules respectively.

>> **foyer_kwargs**
>> [dict, optional, default=None] Keyword arguments to provide to *foyer.Forcefield.apply*. Depending on the file extension these will be

passed to either *write_gsd*, *write_hoomdxml*, *write_lammpsdata*, *write_mcf*, or *parmed.Structure.save*. See parmed structure documentation[70]

parmed_kwargs
[dict, optional, default=None] Keyword arguments to provide to `mbuild.Compound.to_parmed()`

**kwargs
Depending on the file extension these will be passed to either *write_gsd*, *write_hoomdxml*, *write_lammpsdata*, *write_mcf*, or *parmed.Structure.save*. See https://parmed.github.io/ParmEd/html/structobj/parmed.structure. Structure.html#parmed.structure.Structure.save

Other Parameters

ref_distance
[float, optional, default=1.0] Normalization factor used when saving to .gsd and .hoomdxml formats for converting distance values to reduced units.

ref_energy
[float, optional, default=1.0] Normalization factor used when saving to .gsd and .hoomdxml formats for converting energy values to reduced units.

ref_mass
[float, optional, default=1.0] Normalization factor used when saving to .gsd and .hoomdxml formats for converting mass values to reduced units.

atom_style: str, default='full'
Defines the style of atoms to be saved in a LAMMPS data file. The following atom styles are currently supported: 'full', 'atomic', 'charge', 'molecular' See LAMMPS atom style documentation[71] for more information.

unit_style: str, default='real'
Defines to unit style to be save in a LAMMPS data file. Defaults to 'real' units. Current styles are supported: 'real', 'lj'. See LAMMPS unit style documentation_[72] for more information.

See also:

`conversion.save`
Main saver logic

`formats.gsdwrite.write_gsd`
Write to GSD format

`formats.hoomdxml.write_hoomdxml`
Write to Hoomd XML format

`formats.xyzwriter.write_xyz`
Write to XYZ format

`formats.lammpsdata.write_lammpsdata`
Write to LAMMPS data format

`formats.cassandramcf.write_mcf`
Write to Cassandra MCF format

`formats.json_formats.compound_to_json`
Write to a json file

When saving the compound as a json, only the following arguments are used: * filename * include_ports

spin(*theta*, *around*, *anchor=None*)

Rotate Compound in place around an arbitrary vector.

    Parameters

        theta
        [float] The angle by which to rotate the Compound, in radians.

        around
        [np.ndarray, shape=(3,), dtype=float] The axis about which to spin the Compound.

        anchor
        [mb.Compound, optional, default=None (self)] Anchor compound/particle to perform spinning. If the anchor is not a particle, the spin will be around the center of the anchor Compound.

successors()

Yield Compounds below self in the hierarchy.

    Yields

        mb.Compound
        The next Particle below self in the hierarchy

to_gmso(***kwargs*)

Create a GMSO Topology from a mBuild Compound.

    Parameters

        compound
        [mb.Compound] The mb.Compound to be converted.

    Returns

        topology
        [gmso.Topology] The converted gmso Topology

to_intermol(*molecule_types=None*)

Create an InterMol system from a Compound.

    Parameters

        molecule_types
        [list or tuple of subclasses of Compound]

    Returns

        intermol_system
        [intermol.system.System]

    See also:

    `mbuild.conversion.to_intermol`

to_networkx(*names_only=False*)

Create a NetworkX graph representing the hierarchy of a Compound.

    Parameters

names_only

    [bool, optional, default=False] Store only the names of the compounds in the graph, appended with their IDs, for distinction even if they have the same name. When set to False, the default behavior, the nodes are the compounds themselves.

    Returns

    G

        [networkx.DiGraph]

See also:

`mbuild.conversion.to_networkx`
`mbuild.bond_graph`

Notes

This digraph is not the bondgraph of the compound.

`to_parmed`(*box=None, title='', residues=None, include_ports=False, infer_residues=False, infer_residues_kwargs={}*)

Create a ParmEd Structure from a Compound.

    Parameters

    box

    [mb.Box, optional, default=self.boundingbox (with buffer)] Box information to be used when converting to a *Structure*. If 'None', self.box is used. If self.box is None, a bounding box is used with 0.5 nm buffer in each dimension to avoid overlapping atoms.

    title

    [str, optional, default=self.name] Title/name of the ParmEd Structure

    residues

    [str of list of str, optional, default=None] Labels of residues in the Compound. Residues are assigned by checking against Compound.name.

    include_ports

    [boolean, optional, default=False] Include all port atoms when converting to a *Structure*.

    infer_residues

    [bool, optional, default=True] Attempt to assign residues based on the number of bonds and particles in an object. This option is not used if *residues == None*

    infer_residues_kwargs

    [dict, optional, default={}] Keyword arguments for `mbuild.conversion.pull_residues()`

    Returns

    parmed.structure.Structure
    ParmEd Structure object converted from self

See also:

`mbuild.conversion.to_parmed`
`parmed.structure.Structure`
    Details on the ParmEd Structure object

to_pybel(*box=None, title='', residues=None, include_ports=False, infer_residues=False*)

    Create a pybel.Molecule from a Compound.

        Parameters

            box

                [mb.Box, def None]

            title

                [str, optional, default=self.name] Title/name of the ParmEd Structure

            residues

                [str of list of str] Labels of residues in the Compound. Residues are assigned by checking against Compound.name.

            include_ports

                [boolean, optional, default=False] Include all port atoms when converting to a *Structure*.

            infer_residues

                [bool, optional, default=False] Attempt to assign residues based on names of children

        Returns

            pybel.Molecule

    See also:

    `mbuild.conversion.to_pybel`

    Notes

    Most of the mb.Compound is first converted to openbabel.OBMol And then pybel creates a pybel.Molecule from the OBMol Bond orders are assumed to be 1 OBMol atom indexing starts at 1, with spatial dimension Angstrom

to_rdkit()

    Create an RDKit RWMol from an mBuild Compound.

        Returns

            rdkit.Chem.RWmol

    Notes

    Use this method to utilzie rdkit funcitonality. This method only works when the mBuild compound contains accurate element information. As a result, this method is not compatible with compounds containing abstract particles (e.g. coarse-grained systems)

to_smiles(*backend='pybel'*)

    Create a SMILES string from an mbuild compound.

        Parameters

            compound

                [mb.Compound.] The mbuild compound to be converted.

            backend

                [str, optional, default="pybel"] Backend used to do the conversion.

to_trajectory(*include_ports=False*, *chains=None*, *residues=None*, *box=None*)

>   Convert to an md.Trajectory and flatten the compound.

>   **Parameters**

>>   **include_ports**
>>   [bool, optional, default=False] Include all port atoms when converting to trajectory.

>>   **chains**
>>   [mb.Compound or list of mb.Compound] Chain types to add to the topology

>>   **residues**
>>   [str of list of str] Labels of residues in the Compound. Residues are assigned by checking against Compound.name.

>>   **box**
>>   [mb.Box, optional, default=self.boundingbox (with buffer)] Box information to be used when converting to a *Trajectory*. If 'None', self.box is used. If self.box is None, a bounding box is used with a 0.5 nm buffer in each dimension to avoid overlapping atoms.

>   **Returns**

>>   **trajectory**
>>   [md.Trajectory]

>   **See also:**

>   **_to_topology**

translate(*by*)

>   Translate the Compound by a vector.

>   **Parameters**

>>   **by**
>>   [np.ndarray, shape=(3,), dtype=float]

translate_to(*pos*)

>   Translate the Compound to a specific position.

>   **Parameters**

>>   **pos**
>>   [np.ndarray, shape=3(,), dtype=float]

unlabel_rigid_bodies()

>   Remove all rigid body labels from the Compound.

update_coordinates(*filename*, *update_port_locations=True*)

>   Update the coordinates of this Compound from a file.

>   **Parameters**

>>   **filename**
>>   [str] Name of file from which to load coordinates. Supported file types are the same as those supported by load()

>>   **update_port_locations**
>>   [bool, optional, default=True] Update the locations of Ports so that they are shifted along with their anchor particles. Note: This conserves the location of Ports with respect to the anchor Particle, but does not conserve the orientation of Ports with respect to the molecule as a whole.

See also:

load
    Load coordinates from a file

visualize(*show_ports=False*, *backend='py3dmol'*, *color_scheme={}*, *bead_size=0.3*)
    Visualize the Compound using py3dmol (default) or nglview.

    Allows for visualization of a Compound within a Jupyter Notebook.

    Parameters

        show_ports
            [bool, optional, default=False] Visualize Ports in addition to Particles

        backend
            [str, optional, default='py3dmol'] Specify the backend package to visualize compounds Currently supported: py3dmol, nglview

        color_scheme
            [dict, optional] Specify coloring for non-elemental particles keys are strings of the particle names values are strings of the colors i.e. {'_CGBEAD': 'blue'}

        bead_size
            [float, Optional, default=0.3] Size of beads in visualization

property xyz
    Return all particle coordinates in this compound.

    Returns

        pos
            [np.ndarray, shape=(n, 3), dtype=float] Array with the positions of all particles.

property xyz_with_ports
    Return all particle coordinates in this compound including ports.

    Returns

        pos
            [np.ndarray, shape=(n, 3), dtype=float] Array with the positions of all particles and ports.

---

[64] http://openbabel.org/docs/dev/
[65] http://openmm.org/
[66] https://github.com/mosdef-hub/foyer
[67] http://open-babel.readthedocs.io/en/latest/Forcefields/Overview.html
[68] http://docs.openmm.org/7.0.0/userguide/application.html#creating-force-fields
[69] https://github.com/mosdef-hub/foyer/tree/master/foyer/forcefields
[70] https://parmed.github.io/ParmEd/html/structobj/parmed.structure.Structure.html#parmed.structure.Structure.save
[71] https://lammps.sandia.gov/doc/atom_style.html
[72] https://lammps.sandia.gov/doc/units.html

Box

class mbuild.**Box**(*lengths*, *angles=None*, *precision=None*)

A box representing the bounds of the system.

> **Parameters**
>
> > **lengths**
> >
> > [list-like, shape=(3,), dtype=float] Lengths of the edges of the box.
> >
> > **angles**
> >
> > [list-like, shape=(3,), dtype=float, default=None] Angles (in degrees) that define the tilt of the edges of the box. If None is given, angles are assumed to be [90.0, 90.0, 90.0]. These are also known as alpha, beta, gamma in the crystallography community.
> >
> > **precision**
> >
> > [int, optional, default=None] Control the precision of the floating point representation of box attributes. If none provided, the default is 6 decimals.

> **Notes**
>
> Box vectors are expected to be provided in row-major format.
>
> > **Attributes**
> >
> > > *vectors*
> > >
> > > [np.ndarray, shape=(3,3), dtype=float] Box representation as a 3x3 matrix.
> > >
> > > *lengths*
> > >
> > > [tuple, shape=(3,), dtype=float] Lengths of the box.
> > >
> > > *angles*
> > >
> > > [tuple, shape=(3,), dtype=float] Angles defining the tilt of the box (alpha, beta, gamma).
> > >
> > > *Lx*
> > >
> > > [float] Length in the x direction.
> > >
> > > *Ly*
> > >
> > > [float] Length in the y direction.
> > >
> > > *Lz*
> > >
> > > [float] Length in the z direction.
> > >
> > > *xy*
> > >
> > > [float] Tilt factor xy of the box.
> > >
> > > *xz*
> > >
> > > [float] Tilt factor xz of the box.
> > >
> > > *yz*
> > >
> > > [float] Tilt factor yz of the box.
> > >
> > > *precision*
> > >
> > > [int] Amount of decimals to represent floating point values.

> **property Lx**
>
> Length in the x direction.

> **property Ly**
>
> Length in the y direction.

property Lz

Length in the z direction.

property angles

Angles defining the tilt of the box (alpha, beta, gamma).

property box_parameters

Lengths and tilt factors of the box.

property bravais_parameters

Return the Box representation as Bravais lattice parameters.

Based on the box vectors, return the parameters to describe the box in terms of the Bravais lattice parameters:

a,b,c = the edges of the Box alpha, beta, gamma = angles(tilt) of the parallelepiped, in degrees

Returns

parameters
[tuple of floats,] (a, b, c, alpha, beta, gamma)

classmethod from_lengths_angles(*lengths, angles, precision=None*)

Generate a box from lengths and angles.

classmethod from_lengths_tilt_factors(*lengths, tilt_factors=None, precision=None*)

Generate a box from box lengths and tilt factors.

classmethod from_lo_hi_tilt_factors(*lo, hi, tilt_factors, precision=None*)

Generate a box from a lo, hi convention and tilt factors.

classmethod from_mins_maxs_angles(*mins, maxs, angles, precision=None*)

Generate a box from min/max distance calculations and angles.

classmethod from_uvec_lengths(*uvec, lengths, precision=None*)

Generate a box from unit vectors and lengths.

classmethod from_vectors(*vectors, precision=None*)

Generate a box from box vectors.

property lengths

Lengths of the box.

property precision

Amount of decimals to represent floating point values.

property tilt_factors

Return the 3 tilt_factors (xy, xz, yz) of the box.

property vectors

Box representation as a 3x3 matrix.

property xy

Tilt factor xy of the box.

property xz

Tilt factor xz of the box.

property yz

Tilt factor yz of the box.

Lattice

`class` mbuild.**Lattice**(*lattice_spacing=None, lattice_vectors=None, lattice_points=None, angles=None*)

Develop crystal structure from user defined inputs.

Lattice, the abstract building block of a crystal cell. Once defined by the user, the lattice can then be populated with Compounds and replicated as many cell lengths desired in 3D space.

A Lattice is defined through the Bravais lattice definitions. With edge vectors a1, a2, a3; lattice spacing a,b,c; and lattice points at unique fractional positions between 0-1 in 3 dimensions. This encapsulates distance, area, volume, depending on the parameters defined.

>   **Parameters**

>   >   **lattice_spacing**
>   >   >   [array-like, shape=(3,), required, dtype=float] Array of lattice spacings a,b,c for the cell.

>   >   **lattice_vectors**
>   >   >   [array-like, shape=(3, 3), optional, default=None,] Vectors that encase the unit cell corresponding to dimension. Will only default to these values if no angles were defined as well. If None is given, assumes an identity matrix [[1,0,0], [0,1,0], [0,0,1]]

>   >   **lattice_points**
>   >   >   [dictionary, shape={'id': [[nested list of positions]]] optional, default={'default': [[0.,0.,0.]]} Locations of all lattice points in cell using fractional coordinates.

>   >   **angles**
>   >   >   [array-like, shape=(3,), optional, dtype=float] Array of inter-planar Bravais angles in degrees.

Examples

Generating a triclinic lattice for cholesterol.

```
>>> import mbuild as mb
>>> from mbuild.utils.io import get_fn
>>> # reading in the lattice parameters for crystalline cholesterol
>>> angle_values = [94.64, 90.67, 96.32]
>>> spacing = [1.4172, 3.4209, 1.0481]
>>> basis = {'cholesterol':[[0., 0., 0.]]}
>>> cholesterol_lattice = mb.Lattice(spacing,
...                                  angles=angle_values,
...                                  lattice_points=basis)
>>>
>>> # The lattice based on the bravais lattice parameters of crystalline
>>> # cholesterol was generated.
>>>
>>> # Replicating the triclinic unit cell out 3 replications
>>> # in x,y,z directions.
>>>
>>> cholesterol_unit = mb.Compound()
>>> cholesterol_unit = mb.load(get_fn('cholesterol.pdb'))
>>> # associate basis vector with id 'cholesterol' to cholesterol Compound
>>> basis_dictionary = {'cholesterol' : cholesterol_unit}
>>> expanded_cell = cholesterol_lattice.populate(x=3, y=3, z=3,
...                          compound_dict=basis_dictionary)
```

The unit cell of cholesterol was associated with a Compound that contains the connectivity data and spatial arrangements of a cholesterol molecule. The unit cell was then expanded out in x,y,z directions and cholesterol Compounds were populated.

Generating BCC CsCl crystal structure

```
>>> import mbuild as mb
>>> chlorine = mb.Compound(name='Cl')
>>> # angles not needed, when not provided, defaults to 90,90,90
>>> cesium = mb.Compound(name='Cs')
>>> spacing = [.4123, .4123, .4123]
>>> basis = {'Cl' : [[0., 0., 0.]], 'Cs' : [[.5, .5, .5]]}
>>> cscl_lattice = mb.Lattice(spacing, lattice_points=basis)
>>>
>>> # Now associate id with Compounds for lattice points and replicate 3x
>>>
>>> cscl_dict = {'Cl' : chlorine, 'Cs' : cesium}
>>> cscl_compound = cscl_lattice.populate(x=3, y=3, z=3,
...                                       compound_dict=cscl_dict)
```

A multi-Compound basis was created and replicated. For each unique basis atom position, a separate entry must be completed for the basis_atom input.

Generating FCC Copper cell with lattice_vectors instead of angles

```
>>> import mbuild as mb
>>> copper = mb.Compound(name='Cu')
>>> lattice_vector = [[1, 0, 0], [0, 1, 0], [0, 0, 1]]
>>> spacing = [.36149, .36149, .36149]
>>> copper_locations = [[0., 0., 0.], [.5, .5, 0.],
...                     [.5, 0., .5], [0., .5, .5]]
>>> basis = {'Cu' : copper_locations}
>>> copper_lattice = mb.Lattice(lattice_spacing = spacing,
...                             lattice_vectors=lattice_vector,
...                             lattice_points=basis)
>>> copper_dict = {'Cu' : copper}
>>> copper_pillar = copper_lattice.populate(x=3, y=3, z=20,
...                                         compound_dict=copper_dict)
```

Generating the 2d Structure Graphene carbon backbone

```
>>> import mbuild as mb
>>> carbon = mb.Compound(name='C')
>>> angles = [90, 90, 120]
>>> carbon_locations = [[0, 0, 0], [2/3, 1/3, 0]]
>>> basis = {'C' : carbon_locations}
>>> graphene = mb.Lattice(lattice_spacing=[.2456, .2456, 0],
...                       angles=angles, lattice_points=basis)
>>> carbon_dict = {'C' : carbon}
>>> graphene_cell = graphene.populate(compound_dict=carbon_dict,
...                                   x=3, y=3, z=1)
```

Attributes

dimension
    [int, 3] Default dimensionality within mBuild. If choosing a lower dimension, pad the relevant arrays with zeroes.

lattice_spacing
[numpy array, shape=(3,), required, dtype=float] Array of lattice spacings a,b,c for the cell.

lattice_vectors
[numpy array, shape=(3, 3), optional] default=[[1,0,0], [0,1,0], [0,0,1]] Vectors that encase the unit cell corresponding to dimension. Will only default to these values if no angles were defined as well.

lattice_points
[dictionary, shape={'id': [[nested list of positions]]] optional, default={'default': [[0.,0.,0.]]} Locations of all lattice points in cell using fractional coordinates.

angles
[numpy array, shape=(3,), optional, dtype=float] Array of inter-planar Bravais angles

populate(*compound_dict=None, x=1, y=1, z=1*)

Expand lattice and create compound from lattice.

Expands lattice based on user input. The user must also pass in a dictionary that contains the keys that exist in the basis_dict. The corresponding Compound will be the full lattice returned to the user.

If no dictionary is passed to the user, Dummy Compounds will be used.

Parameters

x
[int, optional, default=1] How many iterations in the x direction.

y
[int, optional, default=1] How many iterations in the y direction.

z
[int, optional, default=1] How many iterations in the z direction.

compound_dict
[dictionary, optional, default=None] Link between basis_dict and Compounds.

Raises

ValueError
incorrect x,y, or z values.

TypeError
incorrect type for basis vector

Notes

Called after constructor by user.

Port

class mbuild.**Port**(*anchor=None*, *orientation=None*, *separation=0*)

A set of four ghost Particles used to connect parts.

> **Parameters**
>
>> **anchor**
>>> [mb.Particle, optional, default=None] A Particle associated with the port. Used to form bonds.
>>
>> **orientation**
>>> [array-like, shape=(3,), optional, default=[0, 1, 0]] Vector along which to orient the port
>>
>> **separation**
>>> [float, optional, default=0] Distance to shift port along the orientation vector from the anchor particle position. If no anchor is provided, the port will be shifted from the origin.
>
> **Attributes**
>
>> **anchor**
>>> [mb.Particle, optional, default=None] A Particle associated with the port. Used to form bonds.
>>
>> **up**
>>> [mb.Compound] Collection of 4 ghost particles used to perform equivalence transforms. Faces the opposite direction as self['down'].
>>
>> **down**
>>> [mb.Compound] Collection of 4 ghost particles used to perform equivalence transforms. Faces the opposite direction as self['up'].
>>
>> **used**
>>> [bool] Status of whether a port has been occupied following an equivalence transform.

property access_labels

List labels used to access the Port.

> **Returns**
>
>> **list of str**
>>> Strings that can be used to access this Port relative to self.root

property center

Get the cartesian center of the Port.

property direction

Get the unit vector pointing in the 'direction' of the Port.

property separation

Get the distance between a port and its anchor particle.

If the port has no anchor particle, returns None.

update_orientation(*orientation*)

Change the direction between a port and its anchor particle.

> **orientation**
>> [array-like, shape=(3,), required] Vector along which to orient the port

update_separation(*separation*)

> Change the distance between a port and its anchor particle.

> **separation**
>> [float, required] Distance to shift port along the orientation vector from the anchor particle position. If no anchor is provided, the port will be shifted from the origin.

1.8 Loading Data

Data is input into `mBuild` in a few different ways or from different file types.

Load

Load a file or an existing topology into an mbuild Compound.

Files are read using the predefined backend, unless otherwise specified by the user (through the *backend* flag). Supported backends include "pybel", "mdtraj", "parmed", "rdkit", and "internal". Please refer to http://mdtraj.org/1.8.0/load_functions.html for formats supported by MDTraj and https://parmed.github.io/ParmEd/html/readwrite.html for formats supported by ParmEd.

Parameters

filename_or_object
> [str, mdtraj.Trajectory, parmed.Structure,] mbuild.Compound, pybel.Molecule, Name of the file or topology from which to load atom and bond information.

relative_to_module
> [str, optional, default=None] Instead of looking in the current working directory, look for the file where this module is defined. This is typically used in Compound classes that will be instantiated from a different directory (such as the Compounds located in mbuild.lib).

compound
> [mb.Compound, optional, default=None] Existing compound to load atom and bond information into. New structure will be added to the existing compound as a sub compound.

coords_only
> [bool, optional, default=False] Only load the coordinates into an existing compound.

rigid
> [bool, optional, default=False] Treat the compound as a rigid body

backend
> [str, optional, default=None] Backend used to load structure from file or string. If not specified, a default backend (extension specific) will be used.

smiles: bool, optional, default=False
> Use RDKit or OpenBabel to parse filename as a SMILES string or file containing a SMILES string. If this is set to True, *rdkit* is the default backend.

infer_hierarchy
> [bool, optional, default=True] If True, infer hierarchy from chains and residues

ignore_box_warn
> [bool, optional, default=False] If True, ignore warning if no box is present. Defaults to True when loading from SMILES

**kwargs
> [keyword arguments] Key word arguments passed to mdTraj, GMSO, RDKit, or pybel for loading.

Returns

compound : mb.Compound

Notes

If *smiles* is *True*, either *rdkit* (default) or *pybel* can be used, but RDkit is the only option of these that allows the user to specify a random number seed to reproducibly generate the same starting structure. This is NOT possible with *openbabel*, use *rdkit* if you need control over starting structure's position (recommended).

Load CIF

Load a CifFile object into an mbuild.Lattice.

Parameters

wrap_coords
> [bool, False] Wrap the lattice points back into the 0-1 acceptable coordinates.

Returns

mbuild.Lattice

1.9 Coordinate Transformations

The following utility functions provide mechanisms for spatial transformations for mbuild compounds:

mbuild.coordinate_transform.**force_overlap**(*move_this*, *from_positions*, *to_positions*, *add_bond=True*)

> Move a Compound such that a position overlaps with another.

> Computes an affine transformation that maps the from_positions to the respective to_positions, and applies this transformation to the compound.

> > **Parameters**

> > > **move_this**
> > > > [mb.Compound] The Compound to be moved.

> > > **from_positions**
> > > > [np.ndarray, shape=(n, 3), dtype=float] Original positions.

> > > **to_positions**
> > > > [np.ndarray, shape=(n, 3), dtype=float] New positions.

> > > **add_bond**
> > > > [bool, optional, default=True] If *from_positions* and *to_positions* are *Ports*, create a bond between the two anchor atoms.

mbuild.coordinate_transform.**x_axis_transform**(*compound*, *new_origin=None*, *point_on_x_axis=None*, *point_on_xy_plane=None*)

> Move a compound such that the x-axis lies on specified points.

Parameters

> **compound**
>> [mb.Compound] The compound to move.
>
> **new_origin**
>> [mb.Compound or list-like of size 3, default=[0.0, 0.0, 0.0]] Where to place the new origin of the coordinate system.
>
> **point_on_x_axis**
>> [mb.Compound or list-like of size 3, default=[1, 0, 0]] A point on the new x-axis.
>
> **point_on_xy_plane**
>> [mb.Compound or list-like of size 3, default=[1, 0, 0]] A point on the new xy-plane.

mbuild.coordinate_transform.**y_axis_transform**(*compound*, *new_origin=None*, *point_on_y_axis=None*, *point_on_xy_plane=None*)

Move a compound such that the y-axis lies on specified points.

Parameters

> **compound**
>> [mb.Compound] The compound to move.
>
> **new_origin**
>> [mb.Compound or like-like of size 3, default=[0, 0, 0]] Where to place the new origin of the coordinate system.
>
> **point_on_y_axis**
>> [mb.Compound or list-like of size 3, default=[0, 1, 0]] A point on the new y-axis.
>
> **point_on_xy_plane**
>> [mb.Compound or list-like of size 3, default=[0, 1, 0]] A point on the new xy-plane.

mbuild.coordinate_transform.**z_axis_transform**(*compound*, *new_origin=None*, *point_on_z_axis=None*, *point_on_zx_plane=None*)

Move a compound such that the z-axis lies on specified points.

Parameters

> **compound**
>> [mb.Compound] The compound to move.
>
> **new_origin**
>> [mb.Compound or list-like of size 3, default=[0, 0, 0]] Where to place the new origin of the coordinate system.
>
> **point_on_z_axis**
>> [mb.Compound or list-like of size 3, default=[0, 0, 1]] A point on the new z-axis.
>
> **point_on_zx_plane**
>> [mb.Compound or list-like of size 3, default=[0, 0, 1]] A point on the new xz-plane.

mbuild.compound.Compound.**translate**(*self*, *by*)

Translate the Compound by a vector.

Parameters

by
[np.ndarray, shape=(3,), dtype=float]

`mbuild.compound.Compound.`**`translate_to`**`(`*self*, *pos*`)`

Translate the Compound to a specific position.

Parameters

pos
[np.ndarray, shape=3(,), dtype=float]

`mbuild.compound.Compound.`**`rotate`**`(`*self*, *theta*, *around*`)`

Rotate Compound around an arbitrary vector.

Parameters

theta
[float] The angle by which to rotate the Compound, in radians.

around
[np.ndarray, shape=(3,), dtype=float] The vector about which to rotate the Compound.

`mbuild.compound.Compound.`**`spin`**`(`*self*, *theta*, *around*, *anchor=None*`)`

Rotate Compound in place around an arbitrary vector.

Parameters

theta
[float] The angle by which to rotate the Compound, in radians.

around
[np.ndarray, shape=(3,), dtype=float] The axis about which to spin the Compound.

anchor
[mb.Compound, optional, default=None (self)] Anchor compound/particle to perform spinning. If the anchor is not a particle, the spin will be around the center of the anchor Compound.

1.10 Recipes

Monolayer

`class mbuild.lib.recipes.monolayer.`**`Monolayer`**`(`*surface, chains, fractions=None, backfill=None, pattern=None, tile_x=1, tile_y=1, **kwargs*`)`

A general monolayer recipe.

Parameters

surface
[mb.Compound] Surface on which the monolayer will be built.

chains
[list of mb.Compounds] The chains to be replicated and attached to the surface.

fractions
[list of floats] The fractions of the pattern to be allocated to each chain.

backfill
> [list of mb.Compound, optional, default=None] If there are fewer chains than there are ports on the surface, copies of *backfill* will be used to fill the remaining ports.

pattern
> [mb.Pattern, optional, default=mb.Random2DPattern] An array of planar binding locations. If not provided, the entire surface will be filled with *chain*.

tile_x
> [int, optional, default=1] Number of times to replicate substrate in x-direction.

tile_y
> [int, optional, default=1] Number of times to replicate substrate in y-direction.

Polymer

class mbuild.lib.recipes.polymer.**Polymer**(*monomers=None, end_groups=None*)

> Connect one or more components in a specified sequence.

Notes

There are two different approaches to using the Polymer class to create polymers

1) Pass in already created Compound instances to the monomers and end_groups parameters when creating a Polymer instance:

> You can then call the Polymer.build() method to create a polymer. This approach can be used if the compounds being passed into the Polymer instance already have the ports created, and correct atomic structure to allow for the monomer-monomer and monomer-end group bonds. These compounds are used as-is when creating the polymer chain.

> **Attributes**

> *monomers*
> > [list of mbuild.Compounds] Get the monomers.

> *end_groups*
> > [list of mbuild.Compounds] Get the end groups.

Methods

add_monomer(monomer, indices, separation, port_labels, orientation, replace)	Use to add a monomer compound to Polymer.monomers
add_end_groups(compound, index, separation, orientation, replace)	Use to add an end group compound to Polymer.end_groups
build(n, sequence)	Use to create a single polymer compound. This method uses the compounds created by calling the add_monomer and add_end_group methods.

add_end_groups(*compound*, *index*, *separation=None*, *orientation=None*, *replace=True*, *label='head'*, *duplicate=True*)

Add an mBuild compound to self.end_groups.

End groups will be used to cap the polymer. Call this function for each unique end group compound to be used in the polymer, or call it once with duplicate=True if the head and tail end groups are the same.

Parameters

compound
[mbuild.Compound] A compound of the end group structure

index
[int] The particle index in compound that represents the bonding site between the end group and polymer. You can specify the index of a particle that will be replaced by the polymer bond or that acts as the bonding site. See the *replace* parameter notes.

separation
[float] The bond length (units nm) desired between monomer and end-group.

orientation
[array-like, shape=(3,), default None] Vector along which to orient the port If *replace=True* and *orientation=None*, the orientation of the bond between the particle being removed and the anchor particle is used. Recommended behavior is to leave orientation set to None if you are using *replace=True*.

replace
[Bool, default True] If True, then the particle identified by *index* will be removed and ports are added to the particle it was initially bonded to. Only use *replace=True* in the case that index points to a hydrogen atom bonded to the desired bonding site particle. If False, then the particle identified by *index* will have a port added and no particle is removed from the end group compound.

label
[str, default 'head'] Whether to add the end group to the 'head or 'tail' of the polymer. If *duplicate=True*, *label* is ignored.

duplicate
[Bool, default True] If True, then *compound* is duplicated and added to *Polymer.end_groups* twice. Set to True if you want the same end group compound at the head and tail of the polymer. If that's the case, you only need to call *add_end_groups()* once. If False, *compound* is not duplicated and only one instance of the end group structure is added to *Polymer.end_groups*. You can call *add_end_groups()* a second time to add another end group.

Notes

Refer to the docstring notes of the add_monomer() function for an explanation of the correct way to use the *replace* and *index* parameters.

add_monomer(*compound*, *indices*, *separation=None*, *orientation=[None, None]*, *replace=True*)

Add a Compound to self.monomers.

The monomers will be used to build the polymer. Call this function for each unique monomer to be used in the polymer.

Parameters

compound

[mbuild.Compound] A compound of the individual monomer

indices

[list of int of length 2] The particle indices of compound that represent the polymer bonding sites. You can specify the indices of particles that will be replaced by the polymer bond, or indices of particles that act as the bonding sites. See the 'replace' parameter notes.

separation

[float, units nm] The bond length desired at the monomer-monomer bonding site. (separation / 2) is used to set the length of each port

orientation

[list of array-like, shape=(3,) of length 2,] default=[None, None] Vector along which to orient the port If replace = True, and orientation = None, the orientation of the bond between the particle being removed and the anchor particle is used. Recommended behavior is to leave orientation set to None if you are using replace=True.
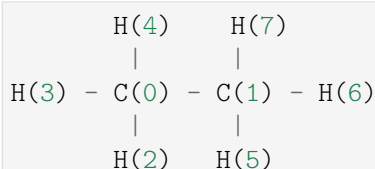
replace

[Bool, required, default=True] If True, then the particles identified by bonding_indices will be removed and ports are added to the particles they were initially bonded to. Only use replace=True in the case that bonding_indices point to hydrogen atoms bonded to the desired monomer-monomer bonding site particles. If False, then the particles identified by bonding_indices will have ports added, and no particles are removed from the monomer compound.

Notes

Using the 'replace' and 'indices' parameters:

The atoms in an mbuild compound can be identified by their index numbers. For example, an ethane compound with the index number next to each atom:

```
      H(4)     H(7)
       |        |
H(3) – C(0) – C(1) – H(6)
       |        |
      H(2)    H(5)
```

If replace=True, then this function removes the hydrogen atoms that are occupying where the C-C bond should occur between monomers. It is required that you specify which atoms should be removed which is achieved by the *indices* parameter.

In this example, you would remove H(2) and H(7) by indicating indices [2, 7]. The resulting structure of the polymer can vary wildly depending on your choice for *indices*, so you will have to test out different combinations to find the two that result in the desired structure.

build(*n*, *sequence='A'*, *add_hydrogens=True*)

Connect one or more components in a specified sequence.

Uses the compounds that are stored in Polymer.monomers and Polymer.end_groups.

If no capping method is used, i.e., if `add_hydrogens == False` and `Polymer.end_groups` is None, then the ports are exposed as, `Polymer.head_port` and `Polymer.tail_port`.

Parameters

n

    [int] The number of times to replicate the sequence.

sequence

    [str, optional, default 'A'] A string of characters where each unique character represents one repetition of a monomer. Characters in *sequence* are assigned to monomers in the order they appear in *Polymer.monomers*. The characters in *sequence* are assigned to the compounds in the in the order that they appear in the Polymer.monomers list. For example, 'AB' where 'A' corresponds to the first compound added to Polymer.monomers and 'B' to the second compound.

add_hydrogens

    [bool, default True] If True and an `end_groups` compound is None, then the head or tail of the polymer will be capped off with hydrogen atoms. If end group compounds exist, then they will be used. If False and an end group compound is None, then the head or tail port will be exposed in the polymer.

periodic_axis

    [str, default None] If not `None` and an `end_groups` compound is None, then the head and tail will be forced into an overlap with a periodicity along the `axis` (default="z") specified. See *mbuild.lib.recipes.polymer.Polymer. create_periodic_bond()* for more details. If end group compounds exist, then there will be a warning. However `add_hydrogens` will simply be overwritten. If `None`, end_groups compound is None, and `add_hydrogens` is False then the head or tail port will be exposed in the polymer.

create_periodic_bond(*axis='z'*)

    Align and bond the end points of a polymer along an axis.

    Parameters

        axis

        [str, default="z"] Axis along which to orient the polymer taken as the line connected the free ports of the end group. May be "x", "y", or "z".

property end_groups

    Get the end groups.

    end_groups cannot be set. Use add_end_group method instead.

property monomers

    Get the monomers.

    monomers cannot be set. Use add_monomer method instead.

Tiled Compound

class mbuild.lib.recipes.tiled_compound.**TiledCompound**(*tile, n_tiles, name=None, **kwargs*)

    Replicates a Compound in any cartesian direction(s).

    Correctly updates connectivity while respecting periodic boundary conditions.

    Parameters

        tile

        [mb.Compound] The Compound to be replicated.

        n_tiles

        [array-like, shape=(3,), dtype=int, optional, default=(1, 1, 1)] Number of times to replicate tile in the x, y and z-directions.

name
> [str, optional, default=tile.name] Descriptive string for the compound.

Silica Interface

class `mbuild.lib.recipes.silica_interface.`**`SilicaInterface`**(*bulk_silica*, *tile_x=1*, *tile_y=1*, *thickness=1.0*, *seed=12345*)

A recipe for creating an interface from bulk silica.

Carves silica interface from bulk, adjusts to a reactive surface site density of 5.0 sites/nm^2 (agreeing with experimental results, see Zhuravlev 2000) by creating Si-O-Si bridges, and yields a 2:1 Si:O ratio (excluding the reactive surface sites).

> **Parameters**
>
> > **bulk_silica**
> > > [Compound] Bulk silica from which to cleave an interface
> >
> > **tile_x**
> > > [int, optional, default=1] Number of times to replicate bulk silica in x-direction
> >
> > **tile_y**
> > > [int, optional, default=1] Number of times to replicate bulk silica in y-direction
> >
> > **thickness**
> > > [float, optional, default=1.0] Thickness of the slab to carve from the silica bulk. (in nm; not including oxygen layers on the top and bottom of the surface)

References

[1], [2]

Packing

mBuild packing module: a wrapper for PACKMOL.

http://leandro.iqm.unicamp.br/m3g/packmol/home.shtml

`mbuild.packing.`**`fill_box`**(*compound*, *n_compounds=None*, *box=None*, *density=None*, *overlap=0.2*, *seed=12345*, *sidemax=100.0*, *edge=0.2*, *compound_ratio=None*, *aspect_ratio=None*, *fix_orientation=False*, *temp_file=None*, *update_port_locations=False*)

Fill a box with an *mbuild.compound* or *Compound* s using PACKMOL.

fill_box takes a single *Compound* or a list of *Compound* s and returns a *Compound* that has been filled to specification by PACKMOL.

When filling a system, two arguments of *n_compounds* , *box* , and *density* must be specified.

If *n_compounds* and *box* are not None, the specified number of compounds will be inserted into a box of the specified size.

If *n_compounds* and *density* are not None, the corresponding box size will be calculated internally. In this case, *n_compounds* must be an int and not a list of int.

If *box* and *density* are not None, the corresponding number of compounds will be calculated internally.

For the cases in which *box* is not specified but generated internally, the default behavior is to calculate a cubic box. Optionally, *aspect_ratio* can be passed to generate a non-cubic box.

Parameters

compound
[mb.Compound or list of mb.Compound] Compound or list of compounds to fill in box.

n_compounds
[int or list of int] Number of compounds to be filled in box.

box
[mb.Box] Box to be filled by compounds.

density
[float, units kg/m^3 , default=None] Target density for the system in macroscale units. If not None, one of *n_compounds* or *box* , but not both, must be specified.

overlap
[float, units nm, default=0.2] Minimum separation between atoms of different molecules.

seed
[int, default=12345] Random seed to be passed to PACKMOL.

sidemax
[float, optional, default=100.0] Needed to build an initial approximation of the molecule distribution in PACKMOL. All system coordinates must fit with in +/- sidemax, so increase sidemax accordingly to your final box size.

edge
[float, units nm, default=0.2] Buffer at the edge of the box to not place molecules. This is necessary in some systems because PACKMOL does not account for periodic boundary conditions in its optimization.

compound_ratio
[list, default=None] Ratio of number of each compound to be put in box. Only used in the case of *density* and *box* having been specified, *n_compounds* not specified, and more than one *compound* .

aspect_ratio
[list of float] If a non-cubic box is desired, the ratio of box lengths in the x, y, and z directions.

fix_orientation
[bool or list of bools] Specify that compounds should not be rotated when filling the box, default=False.

temp_file
[str, default=None] File name to write PACKMOL raw output to.

update_port_locations
[bool, default=False] After packing, port locations can be updated, but since compounds can be rotated, port orientation may be incorrect.

Returns

filled
[mb.Compound]

mbuild.packing.**fill_region**(*compound*, *n_compounds*, *region*, *overlap=0.2*, *bounds=None*, *seed=12345*, *sidemax=100.0*, *edge=0.2*, *fix_orientation=False*, *temp_file=None*, *update_port_locations=False*)

Fill a region of a box with *mbuild.Compound* (s) using PACKMOL.

Parameters

compound
    [mb.Compound or list of mb.Compound] Compound or list of compounds to fill in region.

n_compounds
    [int or list of ints] Number of compounds to be put in region.

region
    [mb.Box or list of mb.Box] Region to be filled by compounds.

overlap
    [float, units nm, default=0.2] Minimum separation between atoms of different molecules.

seed
    [int, default=12345] Random seed to be passed to PACKMOL.

sidemax
    [float, optional, default=100.0] Needed to build an initial approximation of the molecule distribution in PACKMOL. All system coordinates must fit with in +/- sidemax, so increase sidemax accordingly to your final box size.

edge
    [float, units nm, default=0.2] Buffer at the edge of the region to not place molecules. This is necessary in some systems because PACKMOL does not account for periodic boundary conditions in its optimization.

fix_orientation
    [bool or list of bools] Specify that compounds should not be rotated when filling the box, default=False.

bounds
    [list-like of list-likes of floats [[min_x, min_y, min_z, max_x, max_y, max_z], ...], units nm, default=None] Bounding(s) within box to pack compound(s). To pack within a bounding area that is not the full extent of the region, bounds are required. Each item of *compound* must have its own bound specified. Use *None* to indicate a given compound is not bounded, e.g. *[[0., 0., 1., 2., 2., 2.], None]* to bound only the first element of *compound* and not the second.

temp_file
    [str, default=None] File name to write PACKMOL raw output to.

update_port_locations
    [bool, default=False] After packing, port locations can be updated, but since compounds can be rotated, port orientation may be incorrect.

Returns

filled
    [mb.Compound]

If using mulitple regions and compounds, the nth value in each list are used in order.
For example, the third compound will be put in the third region using the third value in n_compounds.

mbuild.packing.**fill_sphere**(*compound*, *sphere*, *n_compounds=None*, *density=None*, *overlap=0.2*, *seed=12345*, *sidemax=100.0*, *edge=0.2*, *compound_ratio=None*, *fix_orientation=False*, *temp_file=None*, *update_port_locations=False*)

Fill a sphere with a compound using PACKMOL.

One argument of *n_compounds and density* must be specified.

If *n_compounds* is not None, the specified number of n_compounds will be inserted into a sphere of the specified size.

If *density* is not None, the corresponding number of compounds will be calculated internally.

Parameters

compound
[mb.Compound or list of mb.Compound] Compound or list of compounds to be put in box.

sphere
[list, units nm] Sphere coordinates in the form [x_center, y_center, z_center, radius]

n_compounds
[int or list of int] Number of compounds to be put in box.

density
[float, units kg/m^3, default=None] Target density for the sphere in macroscale units.

overlap
[float, units nm, default=0.2] Minimum separation between atoms of different molecules.

seed
[int, default=12345] Random seed to be passed to PACKMOL.

sidemax
[float, optional, default=100.0] Needed to build an initial approximation of the molecule distribution in PACKMOL. All system coordinates must fit with in +/- sidemax, so increase sidemax accordingly to your final sphere size

edge
[float, units nm, default=0.2] Buffer at the edge of the sphere to not place molecules. This is necessary in some systems because PACKMOL does not account for periodic boundary conditions in its optimization.

compound_ratio
[list, default=None] Ratio of number of each compound to be put in sphere. Only used in the case of *density* having been specified, *n_compounds* not specified, and more than one *compound*.

fix_orientation
[bool or list of bools] Specify that compounds should not be rotated when filling the sphere, default=False.

temp_file
[str, default=None] File name to write PACKMOL raw output to.

update_port_locations
[bool, default=False] After packing, port locations can be updated, but since compounds can be rotated, port orientation may be incorrect.

Returns

filled
[mb.Compound]

mbuild.packing.**solvate**(*solute, solvent, n_solvent, box, overlap=0.2, seed=12345, sidemax=100.0, edge=0.2, fix_orientation=False, temp_file=None, update_port_locations=False, center_solute=True*)

Solvate a compound in a box of solvent using PACKMOL.

Parameters

> **solute**
>> [mb.Compound] Compound to be placed in a box and solvated.
>
> **solvent**
>> [mb.Compound] Compound to solvate the box.
>
> **n_solvent**
>> [int] Number of solvents to be put in box.
>
> **box**
>> [mb.Box] Box to be filled by compounds.
>
> **overlap**
>> [float, units nm, default=0.2] Minimum separation between atoms of different molecules.
>
> **seed**
>> [int, default=12345] Random seed to be passed to PACKMOL.
>
> **sidemax**
>> [float, optional, default=100.0] Needed to build an initial approximation of the molecule distribution in PACKMOL. All system coordinates must fit with in +/- sidemax, so increase sidemax accordingly to your final box size
>
> **edge**
>> [float, units nm, default=0.2] Buffer at the edge of the box to not place molecules. This is necessary in some systems because PACKMOL does not account for periodic boundary conditions in its optimization.
>
> **fix_orientation**
>> [bool] Specify if solvent should not be rotated when filling box, default=False.
>
> **temp_file**
>> [str, default=None] File name to write PACKMOL raw output to.
>
> **update_port_locations**
>> [bool, default=False] After packing, port locations can be updated, but since compounds can be rotated, port orientation may be incorrect.
>
> **center_solute**
>> [bool, optional, default=True] Move solute center of mass to the center of the *mb.Box* used.

Returns

> **solvated**
>> [mb.Compound]

Pattern

mBuild pattern module.

class mbuild.pattern.**DiskPattern**(*n, **kwargs*)

> Generate N evenly distributed points on the unit circle along z = 0.
>
> Disk is centered at the origin. Algorithm based on Vogel's method.
>
> Code by Alexandre Devert: http://blog.marmakoide.org/?p=1

class mbuild.pattern.**Grid2DPattern**(*n, m, **kwargs*)

> Generate a 2D grid (n x m) of points along z = 0.

Notes

Points span [0,1) along x and y axes

> **Attributes**
>> **n**
>>> [int] Number of grid rows
>>
>> **m**
>>> [int] Number of grid columns

class mbuild.pattern.**Grid3DPattern**(*n, m, l, **kwargs*)

Generate a 3D grid (n x m x l) of points.

Notes

Points span [0,1) along x, y, and z axes

> **Attributes**
>> **n**
>>> [int] Number of grid rows
>>
>> **m**
>>> [int] Number of grid columns
>>
>> **l**
>>> [int] Number of grid aisles

class mbuild.pattern.**Pattern**(*points, orientations=None, scale=None, **kwargs*)

A superclass for molecules spatial Patterns.

Patterns refer to how molecules are arranged either in a box (volume) or 2-D surface. This class could serve as a superclass for different molecules patterns

> **Attributes**
>> **points**
>>> [array or np.array] Positions of molecules in surface or space
>>
>> **orientations**
>>> [dict, optional, default=None] Orientations of ports
>>
>> *scale*
>>> [float, optional, default=None] Scale the points in the Pattern.

apply(*compound, orientation='', compound_port=''*)

Arrange copies of a Compound as specified by the Pattern.

> **Parameters**
>> **compound**
>>> [mb.Compound] mb.Compound to be applied new pattern
>>
>> **orientation**
>>> [dict, optional, default=''] New orientations for ports in compound
>>
>> **compound_port**
>>> [list, optional, default=None] Ports to be applied new orientations
>
> **Returns**
>> **compound**
>>> [mb.Compound] mb.Compound with applied pattern

apply_to_compound(*guest*, *guest_port_name='down'*, *host=None*, *backfill=None*, *backfill_port_name='up'*, *scale=True*)

Attach copies of a guest Compound to Ports on a host Compound.

Parameters

guest
[mb.Compound] The Compound prototype to be applied to the host Compound

guest_port_name
[str, optional, default='down'] The name of the port located on *guest* to attach to the host

host
[mb.Compound, optional, default=None] A Compound with available ports to add copies of *guest* to

backfill
[mb.Compound, optional, default=None] A Compound to add to the remaining available ports on *host* after clones of *guest* have been added for each point in the pattern

backfill_port_name
[str, optional, default='up'] The name of the port located on *backfill* to attach to the host

scale
[bool, optional, default=True] Scale the points in the pattern to the lengths of the *host*'s *boundingbox* and shift them by the hosts mins

Returns

guests
[list of mb.Compound] List of inserted guest compounds on host compound

backfills
[list of mb.Compound] List of inserted backfill compounds on host compound

scale(*by*)

Scale the points in the Pattern.

Parameters

by
[float or np.ndarray, shape=(3,)] The factor to scale by. If a scalar, scale all directions isotropically. If np.ndarray, scale each direction independently

class mbuild.pattern.**Random2DPattern**(*n*, *seed=None*, ***kwargs*)

Generate n random points on a 2D grid along z = 0.

Attributes

n
[int] Number of points to generate

seed
[int] Seed for random number generation

class mbuild.pattern.**Random3DPattern**(*n*, *seed=None*, ***kwargs*)

Generate n random points on a 3D grid.

Attributes

> **n**
>> [int] Number of points to generate
>
> **seed**
>> [int] Seed for random number generation

class `mbuild.pattern.`**`SpherePattern`**(*n, **kwargs*)

> Generate N evenly distributed points on the unit sphere.
>
> Sphere is centered at the origin. Alrgorithm based on the 'Golden Spiral'.
>
> Code by Chris Colbert from the numpy-discussion list: http://mail.scipy.org/pipermail/numpy-discussion/2009-July/043811.html

class `mbuild.pattern.`**`Triangle2DPattern`**(*n, m, shift='n', shift_by=0.5, **kwargs*)

> Generate a 2D triangle (n x m) of points along z = 0.
>
> Generate a square grid of dimensions n by m, then shifts points accordingly to generate a triangular arrangement. shift='n' means shifting occurs in the x direction, where the code shift every other row in the range specified by 'm', and vice versa for shift='m'. By default, shifting will be half the distance betwen neighboring points in the direction of shifting.
>
> ### Notes
>
> Points span [0,1) along x and y axes. This code will allow patterns to be generated that are only periodic in one direction. To generate a periodic pattern, 'm' should be even when shift='n' and vice versa.
>
>> **Attributes**
>>
>> **n**
>>> [int] Number of points along x axis
>>
>> **m**
>>> [int] Number of points along y axis
>>
>> **shift**
>>> [str, optional, default="n"] Allow user to choose the pattern to be shifted by "n" or "m"
>>
>> **shift_by**
>>> [0 <= float <= 1, optional, default=0.5] Normalized distance to be shift the n or m rows, with value between 0 and 1. The value should be the relative distance between two consecutive points defined in shift. shift_by value of 0 will return grid pattern.

1.11 Units

mBuild automatically performs unit conversions in its reader and writer functions. When working with an *mbuild.Compound*, mBuild uses the following units:

Quantity	Units
distance	nm
angle	radians*

* *mbuild.Lattice* and *mbuild.Box* use degrees.

See also foyer unit documentation[73] and ele documentation[74].

1.12 Citing mBuild

If you use mBuild for your research, please cite our paper[75]:

ACS

> Klein, C.; Sallai, J.; Jones, T. J.; Iacovella, C. R.; McCabe, C.; Cummings, P. T. A Hierarchical, Component Based Approach to Screening Properties of Soft Matter. In *Foundations of Molecular Modeling and Simulation. Molecular Modeling and Simulation (Applications and Perspectives)*; Snurr, R. Q., Adjiman, C. S., Kofke, D. A., Eds.; Springer, Singapore, 2016; pp 79-92.

BibTeX

```
@Inbook{Klein2016mBuild,
    author    = "Klein, Christoph and Sallai, János and Jones, Trevor J. and
↪Iacovella, Christopher R. and McCabe, Clare and Cummings, Peter T.",
    editor    = "Snurr, Randall Q and Adjiman, Claire S. and Kofke, David A.",
    title     = "A Hierarchical, Component Based Approach to Screening
↪Properties of Soft Matter",
    bookTitle = "Foundations of Molecular Modeling and Simulation: Select
↪Papers from FOMMS 2015",
    year      = "2016",
    publisher = "Springer Singapore",
    address   = "Singapore",
    pages     = "79--92",
    isbn      = "978-981-10-1128-3",
    doi       = "10.1007/978-981-10-1128-3_5",
    url       = "https://doi.org/10.1007/978-981-10-1128-3_5"
}
```

Download as `BibTeX` or `RIS`

1.13 Older Documentation

Up until mBuild Version 0.10.4[76], the documentation is available as pdf files. Please use the following links to download the documentation as a pdf manual:

- Mbuild Version 0.10.4[77]: Download Here[78]

- Mbuild Version 0.10.3[79]: Download Here[80]

- Mbuild Version 0.10.1[81]: Download Here[82]

- Mbuild Version 0.9.3[83]: Download Here[84]

---

[73] https://foyer.mosdef.org/en/stable/units.html
[74] https://ele-ment.readthedocs.io/en/latest/
[75] http://doi.org/10.1007%2F978-981-10-1128-3_5
[76] https://github.com/mosdef-hub/mbuild/releases/tag/0.10.4
[77] https://github.com/mosdef-hub/mbuild/releases/tag/0.10.4
[78] https://github.com/mosdef-hub/mosdef-hub.github.io/raw/master/old_docs/mbuild.0.10.4.pdf
[79] https://github.com/mosdef-hub/mbuild/releases/tag/0.10.3
[80] https://github.com/mosdef-hub/mosdef-hub.github.io/raw/master/old_docs/mbuild.0.10.3.pdf
[81] https://github.com/mosdef-hub/mbuild/releases/tag/0.10.1
[82] https://github.com/mosdef-hub/mosdef-hub.github.io/raw/master/old_docs/mbuild.0.10.1.pdf
[83] https://github.com/mosdef-hub/mbuild/releases/tag/0.9.3
[84] https://github.com/mosdef-hub/mosdef-hub.github.io/raw/master/old_docs/mbuild.0.9.3.pdf

References

[Eastman2013] P. Eastman, M. S. Friedrichs, J. D. Chodera, R. J. Radmer, C. M. Bruns, J. P. Ku, K. A. Beauchamp, T. J. Lane, L.-P. Wang, D. Shukla, T. Tye, M. Houston, T. Stich, C. Klein, M. R. Shirts, and V. S. Pande. "OpenMM 4: A Reusable, Extensible, Hardware Independent Library for High Performance Molecular Simulation." J. Chem. Theor. Comput. 9(1): 461-469. (2013).

[OBoyle2011] O'Boyle, N.M.; Banck, M.; James, C.A.; Morley, C.; Vandermeersch, T.; Hutchison, G.R. "Open Babel: An open chemical toolbox." (2011) J. Cheminf. 3, 33

[OpenBabel] Open Babel, version X.X.X http://openbabel.org, (installed Month Year)

[Halgren1996a] T.A. Halgren, "Merck molecular force field. I. Basis, form, scope, parameterization, and performance of MMFF94." (1996) J. Comput. Chem. 17, 490-519

[Halgren1996b] T.A. Halgren, "Merck molecular force field. II. MMFF94 van der Waals and electrostatic parameters for intermolecular interactions." (1996) J. Comput. Chem. 17, 520-552

[Halgren1996c] T.A. Halgren, "Merck molecular force field. III. Molecular geometries and vibrational frequencies for MMFF94." (1996) J. Comput. Chem. 17, 553-586

[Halgren1996d] T.A. Halgren and R.B. Nachbar, "Merck molecular force field. IV. Conformational energies and geometries for MMFF94." (1996) J. Comput. Chem. 17, 587-615

[Halgren1996e] T.A. Halgren, "Merck molecular force field. V. Extension of MMFF94 using experimental data, additional computational data, and empirical rules." (1996) J. Comput. Chem. 17, 616-641

[Halgren1999] T.A. Halgren, "MMFF VI. MMFF94s option for energy minimization studies." (1999) J. Comput. Chem. 20, 720-729

[Rappe1992] Rappe, A.K., Casewit, C.J., Colwell, K.S., Goddard, W.A. III, Skiff, W.M. "UFF, a full periodic table force field for molecular mechanics and molecular dynamics simulations." (1992) J. Am. Chem. Soc. 114, 10024-10039

[Wang2004] Wang, J., Wolf, R.M., Caldwell, J.W., Kollman, P.A., Case, D.A. "Development and testing of a general AMBER force field" (2004) J. Comput. Chem. 25, 1157-1174

[Hassinen2001] T. Hassinen and M. Perakyla, "New energy terms for reduced protein models implemented in an off-lattice force field" (2001) J. Comput. Chem. 22, 1229-1242

[1] Hartkamp, R., Siboulet, B., Dufreche, J.-F., Boasne, B. "Ion-specific adsorption and electroosmosis in charged amorphous porous silica." (2015) Phys. Chem. Chem. Phys. 17, 24683-24695

[2] L.T. Zhuravlev, "The surface chemistry of amorphous silica. Zhuravlev model." (2000) Colloids Surf., A. 10, 1-38

Python Module Index

m