mbuild

Release 0.10.3

Mosdef Team

2019-11-27

Contents

1 Installation **2**
- 1.1 Install with conda . . . 2
- 1.2 Install with pip . . . 2
- 1.3 Install an editable version from source . . . 2
- 1.4 Supported Python Versions . . . 3
- 1.5 Testing your installation . . . 3

2 Tutorials **3**
- 2.1 Methane: Compounds and bonds . . . 3
- 2.2 Ethane: Reading from files, Ports and coordinate transforms . . . 5
- 2.3 Monolayer: Complex hierarchies, patterns, tiling and writing to files . . . 7
- 2.4 Point Particles: Basic system initialization . . . 9
- 2.5 Building a Simple Alkane . . . 14

3 Data Structure **18**
- 3.1 Compound . . . 19
- 3.2 Port . . . 32

4 Coordinate transformations **32**

5 Recipes **34**
- 5.1 Monolayer . . . 34
- 5.2 Polymer . . . 34
- 5.3 Tiled Compound . . . 35
- 5.4 Silica Interface . . . 35
- 5.5 Lattice . . . 35
- 5.6 Packing . . . 38
- 5.7 Pattern . . . 41

6 Citing mBuild **43**

References **44**

Python Module Index **45**

mBuild: *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 Installation

1.1 Install with conda[1]

```
$ conda install -c omnia -c mosdef -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 omnia
$ conda config --add channels mosdef
$ conda config --add channels conda-forge
$ conda install mbuild
```

Note: The MDTraj website[2] makes a nice case for using Python and in particular the Anaconda scientific python distribution[3] to manage your numerical and scientific Python packages.

1.2 Install with pip[4]

```
$ pip install mbuild
```

Note: PACKMOL[5] is not available on pip but can be installed from source or via conda.

1.3 Install an editable version from source

```
$ git clone https://github.com/mosdef-hub/mbuild
$ cd mbuild
$ pip install -e .
```

To make your life easier, we recommend that you use a pre-packaged Python distribution like Continuum's Anaconda[6] in order to get all of the dependencies.

[1] http://continuum.io/downloads
[2] http://mdtraj.org/latest/new_to_python.html
[3] http://continuum.io/downloads
[4] https://pypi.org/project/pip/
[5] http://m3g.iqm.unicamp.br/packmol/
[6] https://store.continuum.io/

1.4 Supported Python Versions

Python 3.6 and 3.7 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 3.8 and 3.5 and older, may successfully build and function but no guarantee is made.

1.5 Testing your installation

mBuild uses `py.test` for unit testing. To run them simply type run the following while in the base directory:

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

2 Tutorials

This page was generated from docs/tutorials/tutorial_methane.ipynb[7].

The following section was generated from `docs/tutorials/tutorial_methane.ipynb`

2.1 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:

```
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.

```
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.

[7] https://github.com/mosdef-hub/mbuild/blob/0.10.3/docs/tutorials/tutorial_methane.ipynb

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:

```
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:

```
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]))
```

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

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

..... docs/tutorials/tutorial_methane.ipynb ends here.

This page was generated from docs/tutorials/tutorial_ethane.ipynb[8].

The following section was generated from docs/tutorials/tutorial_ethane.ipynb

2.2 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[9] 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')
mb.translate(ch3, -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`. 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')
mb.translate(ch3, -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.
mb.translate(ch3['up'], [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:

[8] https://github.com/mosdef-hub/mbuild/blob/0.10.3/docs/tutorials/tutorial_ethane.ipynb
[9] http://avogadro.cc/wiki/Main_Page

```
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.

```
import mbuild as mb

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

        mb.load('ch3.pdb', compound=self)
        mb.translate(self, -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.
        mb.translate(self['up'], [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:

```
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']`.

```
ethane.visualize()
```

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

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

```
import mbuild as mb

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

(continues on next page)

(continued from previous page)

```
        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)
```

```

```

..... docs/tutorials/tutorial_ethane.ipynb ends here.

This page was generated from docs/tutorials/tutorial_monolayer.ipynb[10].

The following section was generated from docs/tutorials/tutorial_monolayer.ipynb .....

2.3 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.examples 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__()
```

(continues on next page)

[10] https://github.com/mosdef-hub/mbuild/blob/0.10.3/docs/tutorials/tutorial_monolayer.ipynb

(continued from previous page)

```
        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)
```

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

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

```
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:

```
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:

```
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.

And that's it! Check out `examples.alkane_monolayer` for the fully wrapped class.

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

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

```
```

docs/tutorials/tutorial_monolayer.ipynb ends here.

This page was generated from docs/tutorials/tutorial_simple_LJ.ipynb[11].

The following section was generated from docs/tutorials/tutorial_simple_LJ.ipynb

2.4 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).

```
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

[11] https://github.com/mosdef-hub/mbuild/blob/0.10.3/docs/tutorials/tutorial_simple_LJ.ipynb

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.

```
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]
                    mb.translate(lj_particle, 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).

```
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)
            mb.translate(lj_particle, 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:

```
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)
```

(continues on next page)

(continued from previous page)

```
        for pos in pattern:
            lj_particle = mb.clone(lj_proto)
            mb.translate(lj_particle, 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).

```
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
            mb.translate(lj_particle, 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
            mb.translate(lj_particle, 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.

```
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)
        pattern_sphere.scale(0.5)

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

(continues on next page)

(continued from previous page)

```
            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)
            mb.translate(lj_particle, pos)
            self.add(lj_particle)


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

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

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

```
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(0.5)

        for pos in pattern_sphere:
            lj_particle = mb.clone(lj_proto)
            mb.translate(lj_particle, 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(10)
```

(continues on next page)

(continued from previous page)

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

```
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(1)

        for pos in pattern:
            lj_particle = mb.clone(lj_proto)
            mb.translate(lj_particle, 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(10)
        rnd = random.Random()
        rnd.seed(123)

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

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

..... docs/tutorials/tutorial_simple_LJ.ipynb ends here.

This page was generated from docs/tutorials/tutorial_polymers.ipynb[12].

The following section was generated from docs/tutorials/tutorial_polymers.ipynb .....

2.5 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.

```
import mbuild as mb

class CH2(mb.Compound):
    def __init__(self):
        super(CH2, self).__init__()
        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 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
        mb.translate(self['up'], [0, -0.154/2, 0])
        mb.translate(self['down'], [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.

[12] https://github.com/mosdef-hub/mbuild/blob/0.10.3/docs/tutorials/tutorial_polymers.ipynb

Similarly, the ports should be rotated around the x-axis by the same amount so that atoms can be added in a realistic orientation.

```
import numpy as np
import mbuild as mb

class CH2(mb.Compound):
    def __init__(self):
        super(CH2, self).__init__()
        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[$]')
        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.add(mb.Port(anchor=self[0]), label='up')
        mb.translate(self['up'], [0, -0.154/2, 0])
        mb.rotate(self['up'], theta, around=[1, 0, 0])
        self.add(mb.Port(anchor=self[0]), label='down')
        mb.translate(self['down'], [0, 0.154/2, 0])
        mb.rotate(self['down'], -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 first rotating the 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
```

(continues on next page)

(continued from previous page)

```
class CH2(mb.Compound):
    def __init__(self):
        super(CH2, self).__init__()
        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[$]')
        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.add(mb.Port(anchor=self[0]), label='up')
        mb.translate(self['up'], [0, -0.154/2, 0])
        mb.rotate(self['up'], theta, around=[1, 0, 0])
        self.add(mb.Port(anchor=self[0]), label='down')
        mb.translate(self['down'], [0, 0.154/2, 0])
        mb.rotate(self['down'], np.pi, [0, 1, 0])
        mb.rotate(self['down'], -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
```

```
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 chain length, but also the monomer and the port labels (e.g., 'up' and 'down', since these labels are user defined).

```
polymer = mb.lib.recipes.Polymer(CH2(), 10, port_labels=('up', 'down'))
polymer.visualize()
```

Building a System of Alkanes

A system of alkanes can be constructed by simply cloning the polymer constructed above and translating 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.

```
# create the polymer
polymer = mb.lib.recipes.Polymer(CH2(), 10, port_labels=('up', 'down'))

# 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
mb.rotate(polymer, 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)
    mb.translate(current_polymer, 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

polymer = mb.lib.recipes.Polymer(CH2(), 10, port_labels=('up', 'down'))
system = mb.Compound()
mb.rotate(polymer, 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
        mb.rotate(current_polymer, random.uniform(0, np.pi), around)
    mb.translate(current_polymer, pos)
    system.add(current_polymer)

system.visualize()
```

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

```
polymer = mb.lib.recipes.Polymer(CH2(), 5, port_labels=('up', 'down'))
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.

```
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()
        mb.rotate(last_monomer['down'], random.uniform(-delta,delta), [1, 0, 0])
        mb.rotate(last_monomer['down'], random.uniform(-delta,delta), [0, 1, 0])
        self.add(last_monomer)
        for i in range(chain_length-1):
            current_monomer = mb.clone(monomer_proto)
            mb.rotate(current_monomer['down'], random.uniform(-delta,delta), [1, 0, 0])
            mb.rotate(current_monomer['down'], 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()
```

```
```

docs/tutorials/tutorial_polymers.ipynb ends here.

3 Data Structure

The primary building blocks in an mBuild hierarchy inherit from the `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`. `Ports` are 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 leafs of the tree, are referred to as `Particles` and can be instantiated as `foo = mb.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 `Compound.particles()` traverses the hierarchy to the bottom and yields those `Compounds`. `Compound.root()` returns the compound at the top of the hierarchy.

3.1 Compound

class mbuild.compound.**Compound**(*subcompounds=None*, *name=None*, *pos=None*, *charge=0.0*, *periodicity=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 (Gamma, Erich; Richard Helm; Ralph Johnson; John M. Vlissides (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. 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

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

periodicity [np.ndarray, shape=(3,), dtype=float, optional, default=[0, 0, 0]] The periodic lengths of the Compound in the x, y and z directions. Defaults to zeros which is treated as non-periodic.

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

Attributes

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

children [OrderedSet] 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] The ID of the rigid body that this Compound belongs to. Only Particles (the bottom of the containment hierarchy) can have integer values for *rigid_id*. Compounds containing rigid particles will always have *rigid_id == None*. See also *contains_rigid*.

boundingbox Compute the bounding box of the compound.

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

contains_rigid Returns True if the Compound contains rigid bodies

max_rigid_id Returns the maximum rigid body ID contained in the Compound.

n_particles Return the number of Particles in the Compound.

n_bonds Return the number of bonds in the Compound.

root 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`(*self*, *new_child*, *label=None*, *containment=True*, *replace=False*, *inherit_periodicity=True*, *reset_rigid_ids=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, optional] A descriptive string for the part.

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

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.

`add_bond`(*self*, *particle_pair*)
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

`all_ports`(*self*)
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`(*self*)
Generate all ancestors of the Compound recursively.

Yields

mb.Compound The next Compound above self in the hierarchy

`available_ports`(*self*)
Return all unoccupied Ports referenced by this Compound.

Returns

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

`bonds`(*self*)

Return all bonds in the Compound and sub-Compounds.

Yields

tuple of mb.Compound The next bond in the Compound

See also:

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

`property boundingbox`

Compute the bounding box of the compound.

Returns

mb.Box The bounding box for this Compound

`property center`

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 contains_rigid`

Returns 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.

`energy_minimize`(*self*, *forcefield='UFF'*, *steps=1000*, ***kwargs*)

Perform an energy minimization on a Compound

Default behavior utilizes Open Babel (http://openbabel.org/docs/dev/) to perform an energy minimization/geometry optimization on a Compound by applying a generic force field

Can also utilize OpenMM (http://openmm.org/) to energy minimize after atomtyping a Compound using Foyer (https://github.com/mosdef-hub/foyer) 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', and 'Ghemical'. Please refer to the Open Babel documentation (http://open-babel. readthedocs.io/en/latest/Forcefields/Overview.html) when considering your choice of force field. Utilizing OpenMM for energy minimization requires a forcefield XML file with valid SMARTS strings. Please refer to (http://docs. openmm.org/7.0.0/userguide/application.html#creating-force-fields) for more information.

Keyword Arguments

———

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

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

References

If using _energy_minimize_openmm(), please cite: .. [R92550878d20b-1] 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).

If using _energy_minimize_openbabel(), please cite: .. [R92550878d20b-1] 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

If using the 'MMFF94' force field please also cite the following: .. [R92550878d20b-3] T.A. Halgren, "Merck molecular force field. I. Basis, form,

scope, parameterization, and performance of MMFF94." (1996) J. Comput. Chem. 17, 490-519

If using the 'MMFF94s' force field please cite the above along with: .. [R92550878d20b-8] T.A. Halgren, "MMFF VI. MMFF94s option for energy minimization

studies." (1999) J. Comput. Chem. 20, 720-729

If using the 'UFF' force field please cite the following: .. [R92550878d20b-3] 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

If using the ‘GAFF’ force field please cite the following: .. [R92550878d20b-3] 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

If using the ‘Ghemical’ force field please cite the following: .. [R92550878d20b-3] 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

[R92550878d20b-1], [R92550878d20b-1], [2], [R92550878d20b-3], [4], [5], [6], [7], [R92550878d20b-8], [R92550878d20b-3], [R92550878d20b-3], [R92550878d20b-3]

`from_parmed`(*self*, *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`(*self*, *pybel_mol*, *use_element=True*, *coords_only=False*, *infer_hierarchy=True*)
Create a Compound from a Pybel.Molecule

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

`from_trajectory`(*self*, *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

`generate_bonds`(*self*, *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 between Particles for considering a bond

dmax [float] The maximum distance between Particles for considering a bond

`get_smiles`(*self*)
Get SMILES string for compound

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

Returns

smiles_string: str

`label_rigid_bodies`(*self*, *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,
```

(continues on next page)

(continued from previous page)

```
...                       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 max_rigid_id

Returns 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

min_periodic_distance(*self*, *xyz0*, *xyz1*)

Vectorized distance calculation considering minimum image.

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 n_bonds

Return the number of bonds in the Compound.

Returns

int The number of bonds in the Compound

property n_particles

Return the number of Particles in the Compound.

Returns

int The number of Particles in the Compound

particles(*self*, *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_name`(*self*, *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`(*self*, *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.PeriodicCKDTree, 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.PerioidicCKDTree` mBuild implementation of kd-trees

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

`referenced_ports`(*self*)
Return all Ports referenced by this Compound.

Returns

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

`remove`(*self*, *objs_to_remove*)
Cleanly remove children from the Compound.

Parameters

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

`remove_bond`(*self*, *particle_pair*)
Deletes 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

`rigid_particles`(*self*, *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`
The Compound at the top of self's hierarchy.

Returns

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

`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.

`save`(*self*, *filename*, *show_ports=False*, *forcefield_name=None*, *forcefield_files=None*, *forcefield_debug=False*, *box=None*, *overwrite=False*, *residues=None*, *combining_rule='lorentz'*, *foyer_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 are: 'hoomdxml', 'gsd', 'gro', 'top', 'lammps', 'lmp', 'mcf'

show_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'. Forcefields listed here: https://github.com/mosdef-hub/foyer/tree/master/foyer/forcefields

forcefield_debug [bool, optional, default=False] Choose level of verbosity 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*.

****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 http://lammps.sandia.gov/doc/atom_style.html for more information on atom styles.

See also:

`formats.gsdwrite.write_gsd` Write to GSD format

`formats.hoomdxml.write_hoomdxml` Write to Hoomd XML 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

Notes

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

- filename
- show_ports

`spin`(*self*, *theta*, *around*)

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.

`successors`(*self*)

Yield Compounds below self in the hierarchy.

Yields

mb.Compound The next Particle below self in the hierarchy

`to_intermol`(*self*, *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]

`to_networkx`(*self*, *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.bond_graph`

Notes

This digraph is not the bondgraph of the compound.

`to_parmed`(*self*, *box=None*, *title=''*, *residues=None*, *show_ports=False*, *infer_residues=False*)
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', a bounding box is used with 0.25nm buffers at each face to avoid overlapping atoms, unless *self.periodicity* is not None, in which case those values are used for the box lengths.

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.

show_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

parmed.structure.Structure ParmEd Structure object converted from self

See also:

`parmed.structure.Structure` Details on the ParmEd Structure object

`to_pybel`(*self*, *box=None*, *title=''*, *residues=None*, *show_ports=False*, *infer_residues=False*)
Create a pybel.Molecule from a Compound

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.

show_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

pybel.Molecule

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_trajectory`(*self*, *show_ports=False*, *chains=None*, *residues=None*, *box=None*)
Convert to an md.Trajectory and flatten the compound.

Parameters

show_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', a bounding box is used with a 0.5nm buffer in each dimension. to avoid overlapping atoms, unless *self.periodicity* is not None, in which case those values are used for the box lengths.

Returns

trajectory [md.Trajectory]

See also:

`_to_topology`

`translate`(*self*, *by*)
Translate the Compound by a vector

Parameters

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

`translate_to`(*self*, *pos*)
Translate the Compound to a specific position

Parameters

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

`unlabel_rigid_bodies`(*self*)
Remove all rigid body labels from the Compound

`update_coordinates`(*self*, *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`(*self*, *show_ports=False*, *backend='py3dmol'*, *color_scheme={}*)
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'}

`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.

3.2 Port

class mbuild.port.**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 of 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

The cartesian center of the Port

property direction

The unit vector pointing in the 'direction' of the Port

4 Coordinate transformations

mbuild.coordinate_transform.**force_overlap**(*move_this*, *from_positions*, *to_positions*, *add_bond=True*)

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.**translate**(**args*, ***kwargs*)

mbuild.coordinate_transform.**translate_to**(**args*, ***kwargs*)

mbuild.coordinate_transform.**rotate**(**args*, ***kwargs*)

mbuild.coordinate_transform.**spin**(**args*, ***kwargs*)

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, optional, 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, optional, default=[1.0, 0.0, 0.0]] A point on the new x-axis.

point_on_xy_plane [mb.Compound, or list-like of size 3, optional, default=[1.0, 0.0, 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, optional, default=[0.0, 0.0, 0.0]] Where to place the new origin of the coordinate system.

point_on_y_axis [mb.Compound or list-like of size 3, optional, default=[0.0, 1.0, 0.0]] A point on the new y-axis.

point_on_xy_plane [mb.Compound or list-like of size 3, optional, default=[0.0, 1.0, 0.0]] 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, optional, default=[0.0, 0.0, 0.0]] Where to place the new origin of the coordinate system.

point_on_z_axis [mb.Compound or list-like of size 3, optional, default=[0.0, 0.0, 1.0]] A point on the new z-axis.

point_on_zx_plane [mb.Compound or list-like of size 3, optional, default=[0.0, 0.0, 1.0]] A point on the new xz-plane.

5 Recipes

5.1 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.

5.2 Polymer

class mbuild.lib.recipes.polymer.**Polymer**(*monomers*, *n*, *sequence='A'*, *port_labels=('up', 'down')*)

Connect one or more components in a specified sequence.

Parameters

monomers [mb.Compound or list of mb.Compound] The compound(s) to replicate.

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 assigned by the built-in *sorted()*.

port_labels [2-tuple of strs, optional, default=('up', 'down')] The names of the two ports to use to connect copies of proto.

5.3 Tiled Compound

class mbuild.lib.recipes.tiled_compound.**TiledCompound**(*tile*, *n_tiles*, *name=None*)

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.

5.4 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 [mb.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]

5.5 Lattice

class mbuild.lattice.**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=[[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 [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(*self*, *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.

5.6 Packing

mbuild.packing.**fill_box**(*compound*, *n_compounds=None*, *box=None*, *density=None*, *overlap=0.2*, *seed=12345*, *edge=0.2*, *compound_ratio=None*, *aspect_ratio=None*, *fix_orientation=False*, *temp_file=None*, *update_port_locations=False*)

Fill a box with a *mbuild.compound* or `Compound`s using PACKMOL.

fill_box takes a single *mbuild.Compound* or a list of *mbuild.Compound*'s and return an *mbuild.Compound* that has been filled to the user's specifications to the best of PACKMOL's ability.

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 n_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.

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's 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*, *seed=12345*, *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.

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.

temp_file [str, default=None] File name to write PACKMOL's 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, if 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*, *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.

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's 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*, *edge=0.2*, *fix_orientation=False*, *temp_file=None*, *update_port_locations=False*)

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.

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's 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

solvated [mb.Compound]

5.7 Pattern

`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`(*self*, *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`(*self*, *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 *boundingbox*'s 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`(*self*, *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.`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.`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.`Random2DPattern`(*n*, *seed=None*, ***kwargs*)

`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.`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

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

n [int] Number of grid rows

m [int] Number of grid columns

l [int] Number of grid aisles

6 Citing mBuild

If you use mBuild for your research, please cite our paper[13]:

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`

[13] http://doi.org/10.1007%2F978-981-10-1128-3_5

References

[2] Open Babel, version X.X.X http://openbabel.org, (installed Month Year)

[4] 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

[5] T.A. Halgren, "Merck molecular force field. III. Molecular geometries and vibrational frequencies for MMFF94." (1996) J. Comput. Chem. 17, 553-586

[6] 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

[7] 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

[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

mbuild.packing, 38
mbuild.pattern, 41